Thesis Proposal

Low-level Concurrent Programming Using the Relaxed Memory Calculus

Michael J. Sullivan

December 2015

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.

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 proposal, we suggest that this declarative approach based on explicit programmer-specified constraints is a practical approach for implementing low-level concurrent algorithms. We attempt to establish the plausibility of this thesis by giving a description of programming in C++ using RMC and describing the implementation of a compiler for RMC.

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 [15]. 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 constrain the behavior of programs. The definitions here are very complicated, though, with lots of moving parts.

We propose a much different approach, based on the Relaxed Memory Calculus (RMC) [10]: to have the programmer explicitly specify constraints on the order of execution of operations and on the visibility of memory writes. The compiler then is responsible for ensuring these constraints hold. We believe that the ordering of visibility and execution is the key concept of low-level concurrent programming, and so it is natural to directly expose them in the language. Furthermore, we believe that by having more specific and more fine-grained information about what behavior is allowed, we can generate more efficient code on certain platforms.

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.*

The rest of this proposal is structured as following:

- In Section 2, I discuss the underlying difficulties that present problems for concurrent programming and how some existing languages approach them.
- In Section 3, I present RMC, a new approach to language memory models based on explicit programmer specified constraints.
- In Section 4, I discuss `rmc-compiler`, an LLVM extension for optimizing compilation of C++ and similar languages extended with RMC.
- In Section 5, I discuss the "deliverables" that I intend to use to demonstrate my thesis and present my intended timeline.

Chapter 2

Background

2.1 Sequential Consistency

Sequential consistency is the gold standard for compiling and executing concurrent programs that share memory [15]. 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.

2.2 Paradise Lost

2.2.1 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 [22]. Other architectures, such as POWER and ARM, have memory models so relaxed that most models [2] [21] 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 [12]). 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.

2.2.2 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 somewhat 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:

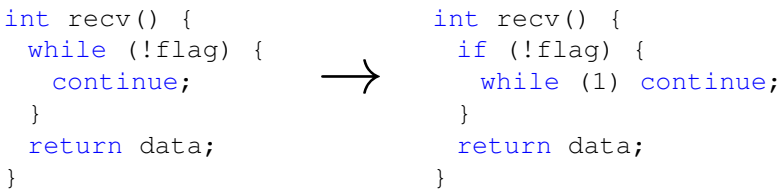
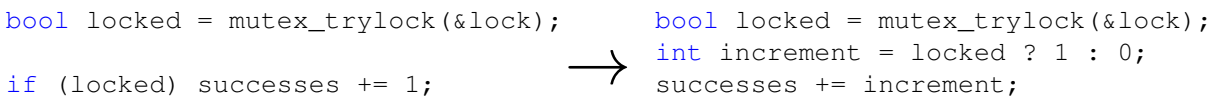


Figure 2.1: Loop hoisting breaking things

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 [24].

2.3 Language Memory Models

Writing lock-free code in a language without a memory model (like C and C++ until recently) is a somewhat 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 [13] [18]. 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 [6].

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 [17] and C++'s [7], 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 2.2.2 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.

2.3.1 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 somewhat 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 [3].

2.3.2 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.

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 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), a 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. McKenney et al. are working to try to repair this situation [20].

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

Chapter 3

The Relaxed Memory Calculus

Much of the material in this section is adapted or copied from "A Calculus for Relaxed Memory" by Karl Crary and myself [10].

3.1 A Tour of RMC

3.1.1 Basics

The Relaxed Memory Calculus (RMC) is a different model for 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:

```
int data, flag;

void send(int msg) {
  data = msg;
  flag = 1;
}
```

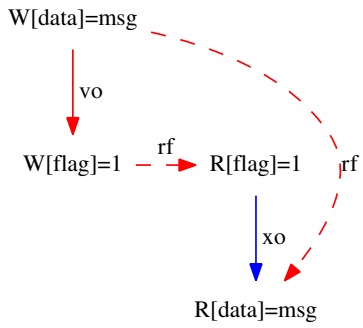
vo

```
int recv() {
  while (!flag)
    continue;
  return data;
}
```

xo

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.

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 ($\xrightarrow{vo}$ and $\xrightarrow{xo}$) are drawn as solid lines while the "reads-from" edges (written $\xrightarrow{rf}$), 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.

3.1.2 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);
}
```

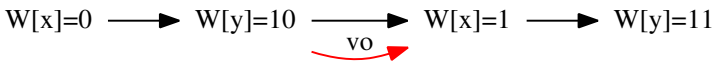
Here, the `L` construct is used to tag expressions. For example, the write `data = msg` is tagged as `wdata` while the read from `flag` is tagged `rflag`. The declaration `VEDGE(wdata, wflag)` creates a visibility-order edge between actions that are tagged `wdata` and actions tagged `wflag`. `XEDGE(rflag,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

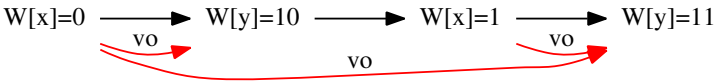
This generates visibility edges from writes to `y` to writes to `x` in future iterations, as shown in this trace (in which unlabeled black lines represent program order):



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 will be important in the ringbuffer example in Section 3.2.1. This behavior can also be at least partially suppressed: calling `rmc_bind_inside()` inside of a function will suppress the generation of edges between invocations of the function. The name comes from its basis in the formalism: the idea is that the labels are being bound *inside* of the function instead of outside of it, so the edges in different invocations of the function refer to different labels.

3.1.3 Pre and post edges

So far, we have showed how to draw fine-grained constraint edges between actions. Sometimes, however, it becomes 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[1] 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 test-and-set that attempts to acquire the lock and with pre-visibility edges to the write

[1] And also currently impossible, as we do not yet have support for constraints between actions in different functions in the implementation.

that releases the lock:

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

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

3.1.4 Transitivity

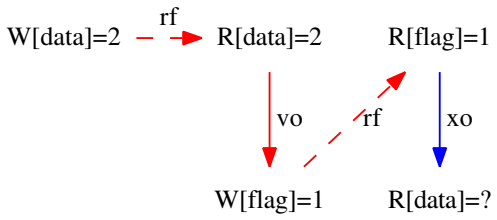
Visibility order and execution order are both transitive. This means that, although the primary meaning of visibility order is in how it relates writes, it is still useful to create edges between other sorts of actions.

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 test-and-set 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.test_and_set()) == 1)
    continue;
  L(acquired, noop());
}
```

Here, the test-and-set 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, which shows a variation on message passing (known as "WWC"):

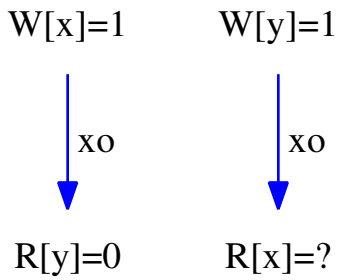


Since reads-from is a form of visibility, and since visibility is transitive, this means that `W[data]=2` is visible before `W[flag]=1`. It is then also visibility ordered before `R[flag]=1`; since

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

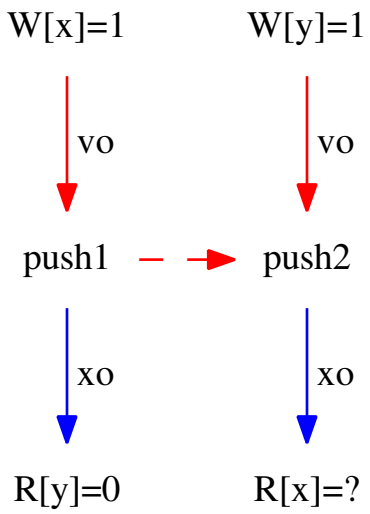
3.1.5 Pushes

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. Somewhat more informally, it means that a will be globally visible before b executes.

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

3.2 Example

3.2.1 Ring-buffers

As a realistic example of code using the RMC memory model, consider the code in Figure 3.2.1. This code—adapted from the Linux kernel [14]—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.

```
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 3.1: 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 many 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 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.)

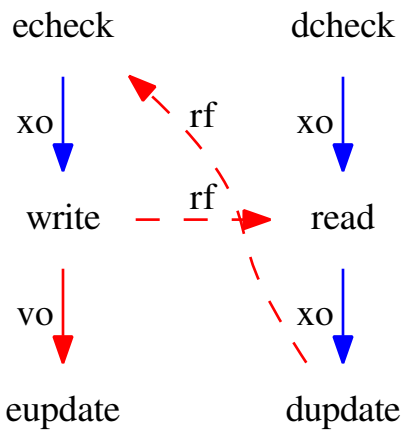


Figure 3.2: Impossible ring-buffer trace

For property (1), the key constraints are `insert` $\overset{vo}{\rightarrow}$ `eupdate` and `dcheck` $\overset{xo}{\rightarrow}$ `read`. If we consider an dequeue reading from some enqueue, `dcheck` reads from `eupdate` and so `insert` $\overset{vo}{\rightarrow}$ `eupdate` $\overset{rf}{\rightarrow}$ `dcheck` $\overset{xo}{\rightarrow}$ `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 3.2. 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` $\overset{xo}{\rightarrow}$ `dupdate` $\overset{rf}{\rightarrow}$ `echeck` $\overset{xo}{\rightarrow}$ `insert` $\overset{rf}{\rightarrow}$ `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.

3.2.2 Using data dependency

One of the biggest complications of the C++11 model is the "consume" memory order, which establishes ordering based on what operations are data dependent. This is useful because it allows read access to data structures to avoid needing any barriers (on ARM and the like) 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) {
  rmc_bind_inside();
  XEDGE(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 `rmc_bind_inside()` is the one concession made in order to enable data dependencies—data dependencies can't really enforce ordering with *all* subsequent invocations 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 mechanism [19].)

3.3 Resolving reads: coherence

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. This order is called *coherence 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.

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, single-threaded programs should work in the obvious way without modification. 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 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 and the operations conflict (that is, they operate on the same location and at least one is a write) (This is crucial for making single threaded programs work properly.)
- A is *visible to* B and they operate on the same location.
- A is transitively ordered before B by the previous two rules.

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

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

3.4 Formalism

RMC is formalized as a core calculus that is used to model possible executions. While we will not present the details of the calculus here, we will sketch some of its key properties. The core calculus is a monadic language with commands for performing memory operations and for creating new labels and edges. Unlike most language memory models, its dynamic semantics are specified operationally and not axiomatically (although the handling of coherence order does have something of an axiomatic flavor, as discussed in Section 3.3). 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.

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`[23], 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 [4]—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.

4.2 x86

While it falls short of sequential consistency, x86's memory model[22] 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 does 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 (somewhat obviously) 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 ARM and POWER

Life is not so simple on ARM and POWER, however. ARM and POWER have a substantially weaker memory model [21] 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.

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 guarantees visibility order. The strictly stronger `sync` instruction on POWER is also sufficient. ARM does not have an equivalent to POWER's `lwsync`, and so we must use the stronger `dmb`, which behaves like `sync`.

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 basically exactly what is needed to implement a push.

While compiling visibility edges and pushes are fairly straightforward and do 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 can 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 can 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 [1] (to produce zero) and then added to an address calculation! And, of course, we can always use the regular barriers.

The different sorts of ways of enforcing execution give 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);
```

[1] At least in previous models. Current ARM [16, §B2.7.4] has a notion of "false" dependencies.

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 somewhat 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.

4.4.2 Compilation Using SMT

We model the problem of enforcing the constraints as an SMT problem and us the Z3 SMT solver [11] to compute the optimal placement of barriers and use of dependencies (according to our metrics). The representation we used was inspired by the integer-linear-programming representation of graph multi-cut [9]—we don't go into detail about modeling our problem as graph multi-cut, since it is not actually particularly more illuminating than the SMT representation is and it does not scale up to handling barriers. 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. [8] 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. [5] 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.

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. The `opt` branch of Z3 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.

When considering the function's CFG, we assume that each labeled action lives in a basic block by itself. Furthermore, unless `rmc_bind_inside()` has been specified for this 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.

As a preprocessing step, we compute the transitive closure of all of the constraint edges for the function (taking into account that visibility implies execution). This is so that we can then safely ignore any edges that don't have any meaning apart from their transitive effects, such as those involving no-ops.

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.

Barrier-only implementation

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

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

Here, the assignments to the lwsync and sync variables produced by the SMT solver are used by the compiler to determine where to insert barriers. We take $\mathsf{paths}(s,t)$ to mean all of the simple paths from s to t. 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:

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

The rules for compiling dealing with push edges is pretty straightforward, with one subtlety. In the RMC description, the "push" *action* is taken as primary, with "push edges" as a convenience to save on typing. In the compiler, since we need to be able to support push edges anyways, we convert explicit pushes into push edges while compiling. This is done by finding all pairs of edges such that $A \xrightarrow{\mathsf{vo}} push \xrightarrow{\mathsf{xo}} B$, discarding the push and the edges, and adding in a

push edge $A \xrightarrow{\mathsf{pu}} B$. The SMT rules for compiling push edges, then, are basically the same as for visibility, except only heavyweight syncs are sufficient to cut an edge:

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

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:

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

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. In the current implementation, the values are made up based on poorly-founded gut feelings, and need to be tuned.

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. Of course, we can't directly enumerate all possible paths between

the actions, since there are infinitely many. But we *do* consider all simple paths—this means that any path from an action A to B that we do not consider must proceed from A to A again, and then to B. We take advantage of this when using dependencies to ensure ordering: if we can use dependencies to order A along all simple paths to B, and can also execution order A with A, then this suffices to order A and B. 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":

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

As discussed above, execution order edges can be cut by barriers, like with visibility, and also by control and data dependencies, if the appropriate side conditions hold.

The control dependency related constraints are somewhat involved:

$$
\begin{aligned}
\mathsf{ctrl}(s,t) &= \bigwedge_{p \in \mathsf{paths}(s,t)} \mathsf{ctrl\_path}(p) \\
\mathsf{ctrl\_path}(p) &= \bigvee_{e \in p} \mathsf{can\_ctrl}(s,e) \wedge \mathsf{use\_ctrl}(s,e) \\
& (\text{where } s = \mathsf{head}(p)) \\
\mathsf{ctrlcut\_path}(p) &= (\mathsf{iswrite}(t) \wedge \mathsf{ctrl\_path}(p)) \vee \mathsf{ctrlisync\_path}(p) \\
& (\text{where } t = \mathsf{tail}(p)) \\
\mathsf{ctrlisync\_path}(p) &= \bigvee_{e::p' \in p} \mathsf{ctrl\_path}(s,p) \wedge \mathsf{isync\_path}(p') \\
& (\text{where } s = \mathsf{head}(p)) \\
\mathsf{isync\_path}(p) &= \bigvee_{e \in p} \mathsf{isync}(e)
\end{aligned}
$$

In this, $\mathsf{can\_ctrl}(s,e)$ 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. $\mathsf{iswrite}(t)$ is also an

input that is true if t is an action containing only writes. 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`.

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

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

Here, $\mathsf{can\_data}(s,v,p)$ 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). The path p needs to be included because whether something is data-dependent can be path-dependent.

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.)

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 $\mathsf{use\_data}$ and $\mathsf{use\_ctrl}$ and summing them up. Different weights should be given based on whether the dependencies are already present or need to be synthesized.

Chapter 5

Proposed Work

To support the thesis that programmer-specified constraints are a practical approach writing lock-free shared memory concurrent programs, I intend to deliver the following:

5.1 An extended Relaxed Memory Calculus

The Relaxed Memory Calculus as described in [10] is an elegant core calculus that embodies the key ideas we wish to use, but it has some important shortcomings that need to be addressed in order to build a practical system on top of it. Optionally, I will extend the existing Coq formalization of the RMC core to include these extensions.

5.1.1 Non-atomic memory locations

The most pressing shortcoming that needs to be addressed is that while RMC was carefully designed to be more permissive than all existing (and most conceivable) architectures, it is actually too restrictive to implement a compiler for without sacrificing performance. The root of this problem is that while RMC is extremely permissive about how memory operations may be reordered and resolved, some of its assumptions look more like a architectural memory model than a language one. One such assumption 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 assumption that writes will eventually propagate out to all threads: as demonstrated in Figure 2.1, 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. In order to address this, I intend to adapt the solution that C++11 uses to the setting of RMC: introduce a notion of "non-atomic" memory locations which are not allowed to be accessed concurrently, on pain of invoking undefined behavior (halt-and-catch-fire). 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 dif-

ferent 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.

To support this, RMC will need to be extended with semantics for non-atomic memory operations and with an appropriate definition of "data race". The implementation already uses this strategy (as can be seen in the examples), but it is not backed by support in the theory.

5.1.2 Sequentially consistent operations

As discussed in Section 2.3.2, 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 3.1.3 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, store);
  L(store, *p = val);
}

int sc_load(rmc::atomic<int> *p) {
  PEDGE(pre, load);
  XEDGE(load, post);
  return L(load, *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 [10], 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, somewhat oddly, 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 in RMC, but all have similar problems.

Atomic operations with a sequentially consistent ordering between them are a useful tool that RMC ought to have an efficient story for. To support this, I will extend RMC with some form of

support for these.

5.2 Compiler improvements

While the compiler is mostly functional, some additional work is still required:

5.2.1 ARMv8 support

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 [16]. 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: "Store-Releases" become visible before any subsequent "Load-Acquires" execute and "Store-Releases" become visible to all other CPUs at the same time (this second property is actually stronger than C++ SC atomics).

While the compiler supports ARM, it does not yet take advantage of the new atomic instructions introduced by ARMv8. The new operations are powerful primitives, and are likely to be well optimized in new processors. Being able to take advantage of these operations would be a good demonstration of the generality of our approach.

I believe ARMv8's "Load-Acquire" should be able to serve as a one-to-many execution edge from a load to everything after it, while "Store-Release" should be able to serve as a many-to-one visibility edge to a store from everything before it. Once sequentially consistent operations are added as discussed in Section 5.1.2, these instructions can be used to implement them. Additionally, it may be possible in some circumstances to use these instructions to implement pushes.

5.2.2 More support for inter-function constraints

The major functional improvement that needs to be made is some 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.

One example of this is the code presented in 3.2.2. 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.

There are a few potential approaches to this problem. One is to simply allow some form of explicit cross-function edges, as shown in that example. Another is to allow labels to be treated as first class objects and passed to functions as arguments. These solutions have the advantage of generality but are difficult to generate good code for if the edges can cross compilation unit boundaries. A less general but perhaps more practical solution is to more narrowly focus on the use-case where this is important (when the cross-function edges are expected to be realized through a data dependency) and provide a mechanism for specifying constraints to and from the "action" of passing an argument or returning a value.

5.2.3 Generalize `rmc_bind_inside()`

The `rmc_bind_inside()` directive provides a way to have more fine-grained control of exactly which executions of a action an edge applies to. This "more fine-grained control", however, is still very coarse-grained: it is only able to suppress edges between invocations of a function, and it is all-or-nothing, applying to all labels and edges in a function or none of them. This is somewhat unsatisfactory, and it shouldn't be hard to do something more fine-grained.

5.3 RMC case studies

To evaluate the suitability of RMC as a methodology for implementing low-level concurrent programs, I intend to implement a collection of concurrent algorithms and data structures using it and provide informal arguments for their correctness. For comparison, I will implement many of the algorithms using C++11 as well. Some of the case studies I intend to perform are:

- Higher-level concurrency primitives, including mutexes (more sophisticated ones than those in Section 3.1.3), readers-writer locks, and condition variables
- Lock-free data structures, including queues and stacks
- A read-copy-update implementation and some RCU client code

5.4 Performance measurements and optimizations

The suite of example programs will also be used as a benchmark suite to evaluate the performance of code generated by `rmc-compiler`. We expect that we can essentially always match the

performance of C++11 style code and can in many cases exceed it (at least on ARM; x86 requires so little work to ensure ordering that there is not much room for improvement).

While the compiler attempts to optimize in its decisions for how to realize edges, the weightings that it uses are based more on gut intuition and guesswork than on any sort of principled measurement. Some of these intuitions are that fewer barriers are slower than more barriers, that inserting control dependencies (with or without an `isync`) should be faster than barriers (because it is purely processor local and doesn't involve communication), and that inserting data dependencies should be better than inserting control dependencies (since it can't cause branch prediction failure or the like).

These gut intuitions, however, don't seem to match reality. I conducted some quick-and-dirty benchmarks of the ring buffer from Section 3.2.1, using different combinations of approaches for implementing execution edges. While the above list of intuitions would imply that inserting a bogus data dependency from `read` to `dupdate` in `buf_dequeue` is the best strategy, this performed *much* worse than simply inserting a `dmb` on the ARM machines I tested this on.

Since these gut intuitions are so suspect, one of the goals of this thesis will be to characterize the performance properties of different methods of ensuring correct ordering. This information will be used to tune the compilation strategy.

5.5 Timeline

The following is my proposed timeline for completing the work I have proposed. I plan to complete the proposed thesis by **December 2016**.

- **March 2016:** Finish implementing extensions to the compiler (Section 5.2)
- **June 2016:** Complete extensions to the theory (Section 5.1)
- **September 2016:** Done with building and benchmarking example programs (Section 5.3) and making necessary compiler improvements
- **December 2016:** Thesis defense

Additionally, I intend to write and submit a conference paper about the implementation of the compiler and the use of RMC in a real language.

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. 4.4.2

[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. 2.2.1

[3] David Aspinall and Jaroslav Ševčík. Java memory model examples: Good, bad and ugly. In *VAMP 2007*, 2007. 2.3.1

[4] 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

[5] 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. 4.4.2

[6] Hans-J. Boehm. Threads cannot be implemented as a library. In *2005 SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005. 2.3

[7] 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. 2.3

[8] 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. 4.4.2

[9] Marie-Christine Costa, Lucas Létocart, and Frédéric Roupin. Minimal multicut and maximal integer multiflow: a survey. *European Journal of Operational Research*, 2005. 4.4.2

[10] Karl Crary and Michael J. Sullivan. A calculus for relaxed memory. In *2015 ACM Symposium on Principles of Programming Languages*, Mumbai, India, January 2015. 1, 3, 5.1, 5.1.2

[11] 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. 4.4.2

[12] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. Tech-

nical report, 1965. 2.2.1

[13] David Howells and Paul E. McKenney. Linux kernel memory barriers. https://www.kernel.org/doc/Documentation/memory-barriers.txt, 2006. 2.3

[14] David Howells and Paul E. McKenney. Circular buffers. https://www.kernel.org/doc/Documentation/circular-buffers.txt, 2010. 3.2.1

[15] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979. 1, 2.1

[16] ARM Ltd. Arm architecture reference manual (armv8, for armv8-a architecture profile), 2014. 1, 5.2.1

[17] 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. 2.3

[18] Paul E. McKenney. Proper care and feeding of return values from rcu_dereference(). https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt, 2014. 2.3

[19] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency. In *Parallel and Distributed Computing and Systems*, 1998. 3.2.2

[20] Paul E. McKenney, Torvald Riegel, and Jeff Preshing. Toward implementation and use of memory_order_consume. C++ standards committee paper WG21/P0098R0,http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0098r0.pdf, 2015. 2.3.2

[21] 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. 2.2.1, 4.3

[22] 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. 2.2.1, 4.2

[23] Michael J. Sullivan. rmc-compiler. https://github.com/msullivan/rmc-compiler, 2015. 4

[24] Ian Lance Taylor. Single threaded memory model. http://www.airs.com/blog/archives/79, 2007. 2.2.2