Compiling a Calculus for Relaxed Memory

Practical constraint-based low-level concurrency

November 2017

Michael J. Sullivan
Carnegie Mellon University
mjsulliv@cs.cmu.edu

Karl Crary
Carnegie Mellon University
crary@cs.cmu.edu

Salil Joshi
AlphaGrep Securities
salil.ssj@gmail.com

Abstract

Crary and Sullivan's Relaxed Memory Calculus (RMC) proposed a new declarative approach for writing low-level shared memory concurrent programs in the presence of modern relaxed-memory multi-processor architectures and optimizing compilers. In RMC, the programmer explicitly specifies constraints on the order of execution of operations and on the visibility of memory writes. These constraints are then enforced by the compiler, which has a wide degree of latitude in how to accomplish its goals.

We present `rmc-compiler`, a Clang and LLVM-based compiler for RMC-extended C and C++. In addition to using barriers to enforce ordering, `rmc-compiler` can take advantage of control and data dependencies, something that is beyond the abilities of current C/C++ compilers. In `rmc-compiler`, RMC compilation is modeled as an SMT problem with a cost term; the solution with the minimum cost determines the compilation strategy. In testing on ARM and POWER devices, RMC performs quite well, with modest performance improvements relative to C++11 on most of our data structure benchmarks and (on some architectures) dramatic improvements on a read-mostly list test that heavily benefits from use of data dependencies for ordering.

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 [17]. Sequential consistency is easy to understand and supports some straightforward reasoning principles, though the exponential growth of the state-space of concurrent programs leaves things still quite tricky.

Modern multi-processor architectures, however, do not provide sequential consistency. Many processors execute instructions out-of-order that are observable from other processors while preserving the behavior of single-threaded computations. More frighteningly, memory subsystems with hierarchical caches and store buffers can themselves propagate writes out of order and even to different processors in different orders. Not to be outdone, compilers get in on the sequential-consistency-violating fun as well: many transformations that are perfectly sensible in a single-threaded setting violate sequential consistency. Excitingly, this includes some bread-and-butter optimizations like common subexpression elimination and loop invariant code motion.

Because the compiler is intimately involved in the problem and because it is preferable to abstract away from the differences between architectures, this is a language design issue [7]. The now-standard language approach to this problem, then, is for languages to guarantee 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. This may not be good enough, however, for performance-critical code and library implementation, requiring languages that target these domains to provide a well defined low-level mechanism for shared memory concurrency. C and C++ (since the C11 and C++11 standards) provide a mechanism based around specifying "memory orderings" when accessing concurrently modified locations [16]. These memory orderings induce constraints that constrain the behavior of programs. While some fragments are fairly pleasant to use, the system as a whole is quite complex and contains many moving parts.

Crary and Sullivan's Relaxed Memory Calculus (RMC) proposes a more declarative new approach to handling weak memory in low-level concurrent programming: explicit, programmer-specified constraints [11]. In RMC, the programmer explicitly specifies constraints on the order of execution of operations and on the visibility of memory writes. 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.

We present a compiler for RMC. The paper is organized as follows:

- In Section 2 we recapitulate the design of RMC in the form of a tutorial introduction to C++ programming using RMC with `rmc-compiler`.
- We present a new informal model (Section 3) of the execution of RMC programs. We detail the split into an *execution model* (Section 3.1) that models the potentially out-of-order execution of program instructions

and a *memory system model* (Section 3.2) that determines what values are read by memory reads.
- We discuss `rmc-compiler`, our LLVM-based compiler for RMC-extended languages and the compilation of RMC to x86, ARM, and POWER (Section 4).
- We detail the potential and difficulties of optimizing barrier placement for RMC programs (Section 5.1) and our modeling of the problem as an SMT problem (Section 5.2).
- We evaluate the performance of RMC programs (Section 6).

2 Programming with RMC

The Relaxed Memory Calculus (RMC) is a declarative approach to handling memory ordering in low-level lock-free concurrent programming. In RMC, 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* ($\xrightarrow{vo}$) and *execution-order* ($\xrightarrow{xo}$). To see the intended meaning of these relations, consider this pair of simple functions for passing a message between two threads:

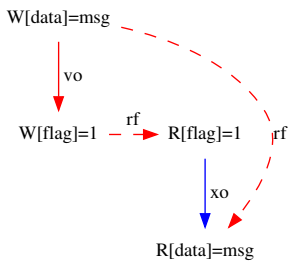
```
int data, flag;

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

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

vo

xo

The visibility edge ($\xrightarrow{vo}$) 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 ($\xrightarrow{xo}$) 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 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.

2.1 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);
}
```

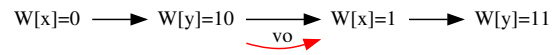
Here, the `L` construct is used to tag expressions. For example, the write `data = msg` is tagged as `wdata` while the read from `flag` is tagged `rflag`. The declaration `VEDGE(wdata, wflag)` creates a visibility-order edge between actions that are tagged `wdata` and actions tagged `wflag`. `XEDGE(rflag,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

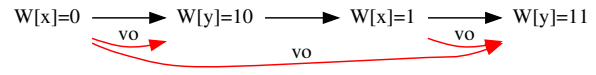
This generates visibility edges from writes to `y` to writes to `x` in future iterations, as shown in this trace (in which unlabeled black lines represent program order):



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:

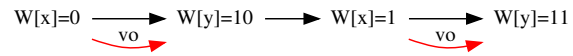


In addition to the obvious visibility edges between writes in the same loop iteration, we also have an edge from the write to `x` in the first iteration to the write to `y` in the second. This behavior will be important in the ring buffer example in Section 6.

While this behavior is a good default, it is sometimes necessary to have more fine-grained 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[1]. 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:



2.2 Pre and post edges

We have discussed drawing 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 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:

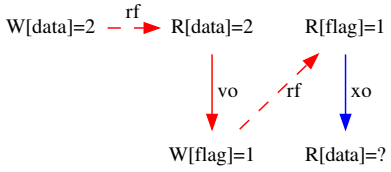
```
void spinlock_lock(spinlock_t *lock) {
  XEDGE(trylock, post);
  while (L(trylock, lock->state.exchange(1)) == 1)
    continue;
}

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

Pre and post edges are a critical piece of the RMC design: the standard pattern for implementing concurrency objects in RMC is to use point-to-point edges for internal constraints and pre/post edges for interfacing with client code.

2.3 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.

The primary substantive application of transitivity occurs with visibility edges from reads to writes. Consider the following trace, which shows a variation on message passing (known as "WWC"):

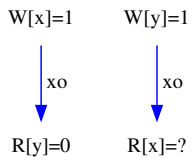


Since reads-from is a form of visibility, and since visibility is transitive, this means that `W[data]=2` is visible before `W[flag]=1`. It is then also visibility ordered before `R[flag]=1`; since that must execute before `R[data]=?`, this means that `W[data]=2` must be *visible to* `R[data]=?`, which will then read from it.

[1] Where the "scope" of a constraint is defined (somewhat unusually) as everything that is dominated by the constraint declaration in the control flow graph.

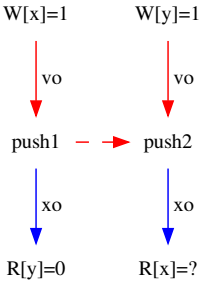
2.4 Pushes

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, the primary means of inserting a push is via a "push edge":

```
PEDGE(write1, read1);
L(write1, x = 1);
r1 = L(read1, y);
```

```
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 1. A ring buffer

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.

2.5 An example: ring-buffers

As a realistic example of code using the RMC memory model, consider the code in Figure 1. This code—adapted from the Linux kernel [15]—implements a ring-buffer, a common data structure that implements an 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.

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

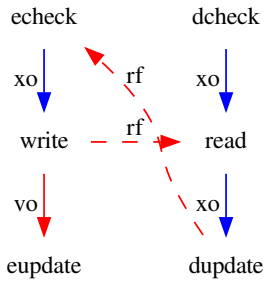


Figure 2. Impossible ring-buffer trace

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.)

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 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.

2.6 Using data dependency

One of the biggest complications of the C++11 model is the "consume" memory order, which establishes ordering

```
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, r);

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

Figure 3. A widget storing library

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.

In Figure 3 we show 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 key to taking advantage of 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.

In order to support this sort of idiom when the edges extend beyond one function, RMC supports drawing edges that involve the "action" of passing a value to a function or returning it as an argument, which provides a way to specify fine-grained ordering constraints between functions. The details of this are omitted for space reasons.

(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 [20].)

3 Model Details

We build on top of a draft version of RMC 2.0 [12], which reworks the formalization of out-of-order execution and rules out out-of-thin-air executions.

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.

3.1 Execution Model

In RMC, the responsibility of ensuring that single-threaded programs behave as expected falls on the memory system model, which allows for an extremely permissive execution model. Intuitively, our execution model 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.

3.2 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. 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. A read, then, can read from any write executed before it such that the constraints on coherence order are obeyed.

As one goal of RMC's is 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 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 is *prior to* C and C is *prior to* B, for some C.

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.

4 Compiling RMC

RMC is implemented by `rmc-compiler` [26], an LLVM plugin 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. C and C++ are implemented using macro libraries that expand to the specifications expected by `rmc-compiler`.

A traditional way to describe how to compile language concurrency constructs is to give direct mappings from language constructs to sequences of assembly instructions [3, 4, 23]. 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.

4.1 x86

While it falls short of sequential consistency, x86's memory model [25] is a pleasure to deal with. On x86, execution

and visibility order come for free—we simply need to prevent the *compiler* from reordering actions. Pushes can be implemented with an `MFENCE` or a locked instruction to some arbitrary location.

4.2 ARM and POWER

Life is not so simple on ARM and POWER, however. POWER has a substantially weaker memory model [24] 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 [2] 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 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.

5 Optimizing RMC Compilation

5.1 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 dmbs before wc and wd. A somewhat more clever implementation might insert dmbs 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.

5.2 Compilation Using SMT

We model the problem of enforcing the constraints as an SMT problem and use the Z3 SMT solver [13] to compute the optimal placement of barriers and use of dependencies (according to our metrics). The representation we use was inspired by an integer-linear-programming representation of graph multi-cut [10]—we don't go into detail about modeling our problem as graph multi-cut, since it is no more illuminating than the SMT representation and does not scale up to using dependencies. This origin survives in our use of the word "cut" to mean satisfying a constraint edge.

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.

When considering the function's CFG, we assume that each labeled action lives in a basic block by itself. Furthermore, 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). 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.

5.2.1 Barrier-only implementation

The rules for encoding the compilation of visibility edges as an SMT problem are reasonably straightforward (for space and simplicity we omit the potential use of `dmb st`):

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

We write $\text{foo}(x)$ to mean a variable in the SMT problem that is given a definition by our equations and $\overline{\text{foo}}(x)$ to mean an "output" variable whose value will be used to drive compilation. Later in this section, 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 $\overline{\text{lwsync}}$ and $\overline{\text{sync}}$ variables produced by the SMT solver are used by the compiler to determine where to insert barriers. We write $s \xrightarrow{\text{vo}} t$ to quantify over visibility edges from s to t and take $\text{paths}(s,t)$ 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 (writing $s \xrightarrow{\text{xo}} t$ to quantify over execution edges):

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

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

$s \xrightarrow{\text{pu}} t$ to quantify over push edges):

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

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:

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

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 $cost_{\text{lwsync}}$ and $cost_{\text{sync}}$ terms are weights representing the costs of the `lwsync` and `sync` instructions, and should be based on their relative costs.

5.2.2 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:

```
rmc::atomic<int> x, y;
void f(bool b) {
  XEDGE(ra, wb);
  int i = L(ra, x);
  if (b) return;
  if (i == 0) {
    L(wb, y = 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.

5.2.3 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":

$$\begin{aligned}
&\bigwedge_{s \xrightarrow{\text{xo}} t} \text{xcut}(s,t) \\
\text{xcut}(s,t) &= \bigwedge_{p \in \text{paths}(s,t)} \text{xcut_path}(p) \\
\text{xcut_path}(p) &= \text{vcut_path}(p) \vee \\
&\quad (\text{ctrlcut_path}(p) \wedge \\
&\quad\ (\text{ctrl}(s,s) \vee \text{xcut}(s,s))) \vee \\
&\quad (\text{datacut_path}(p) \wedge \text{xcut}(s,s)) \\
&\quad (\text{where } s = \text{head}(p))
\end{aligned}$$

As discussed above, execution order edges can be cut along a path by barriers as with visibility (vcut_path), and also by control (ctrlcut_path) and data (datacut_path) dependencies, if the appropriate side conditions hold.

Then, the rules for cutting edges using control (for simplicity, and because they don't help much on ARM, we leave out the rules for using `isync`):

$$\begin{aligned}
\text{ctrl}(s,t) &= \bigwedge_{p \in \text{paths}(s,t)} \text{ctrl_path}(p) \\
\text{ctrl_path}(p) &= \bigvee_{e \in p} \mathit{can_ctrl}(s,e) \wedge \overline{\text{use_ctrl}}(s,e) \\
&\quad (\text{where } s = \text{head}(p)) \\
\text{ctrlcut_path}(p) &= \mathit{iswrite}(t) \wedge \text{ctrl_path}(p) \\
&\quad (\text{where } t = \text{tail}(p))
\end{aligned}$$

In this, $\mathit{can\_ctrl}(s,e)$ 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 and $\mathit{iswrite}(t)$ is true if t is an action containing only writes. Then $\overline{\text{use\_ctrl}}(s,e)$ is the corresponding output, indicating whether the branch will be for ordering purposes.

Here, we can cut an execution edge along a path using control dependencies if the destination of the edge is a write and somewhere along the path there is a branch that is dependent on the source value. This could be extended to support using `isync` regardless of what sort of action the destination is by requiring an `isync` along the path after the branch.

Data dependency is quite simple on the SMT side of things:

$$\begin{aligned}\text{datacut\_path}(p) &= \mathit{can\_data}(s,t,p) \wedge \overline{\text{use\_data}}(s,t,p) \\ &\quad (\text{where } s = \text{head}(p), t = \text{tail}(p))\end{aligned}$$

Here, $\mathit{can\_data}(s, v, p)$ 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 (in LLVM's SSA based intermediate representation, this idea is made explicit through the use of phi nodes).

This check is somewhat subtle: the paths we are trying to cut are only simple paths, but actual execution can follow complex paths (ones with cycles). Thus, we actually must check that a data dependency exists not just when following the simple path p, but also when following any path that "detours" away from p and then returns to it. The data dependence check, then, works by tracing backwards the chain of instruction uses from the destination, looking for the source. When an operand in this chain could come from multiple different basic blocks (via an SSA phi node), we do the check for every source that could have been reached while executing along path p while possibly taking detours off of it.

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 $\overline{\text{use\_data}}$ and $\overline{\text{use\_ctrl}}$ and summing them up. Different weights should be given based on whether the dependencies are already present or need to be synthesized. Currently we use values for the weights that we find work well in practice, though they have not yet been carefully optimized or derived in any particularly principled way.

5.2.4 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:

$$\begin{aligned}\bigwedge_{s \xrightarrow{\text{vo}} t@b} \text{vcut}(b,s,t) \\ \text{vcut}(b,s,t) &= \bigwedge_{p \in \text{paths\_wo}(b,s,t)} \text{vcut\_path}(p)\end{aligned}$$

Here we write $s \longrightarrow t@b$ to indicate a constraint edge from s to t that is bound at b and $\text{paths\_wo}(b, s, t)$ 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.

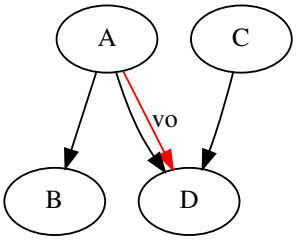
The modifications for execution edges are similar but have one additional wrinkle: when using control and data dependencies to ensure ordering, xcut_path can appeal to xcut and ctrl; we modify xcut_path to pass the binding site down to these. (In fact, this is the main use-case of scoped constraints: 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 $\mathit{can\_data}$ and $\overline{\text{use\_data}}$.

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

5.3 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:



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. By requiring critical edges to be broken, 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 somewhat 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.

5.4 ARMv8

ARMv8 brings with it a number of additions to the ARM memory model that we can take advantage of [2].

ARMv8 adds the `LDA` and `STL` families of instructions, which they name "Load-Acquire" and "Store-Release". Designed to efficient implement C++11 SC atomics, these instructions may be used to realize execution and visibility edges. In RMC terms, Store-Release writes become visible after all program-order prior stores and all stores observed by program-order prior loads, and so are visibility-after all program order predecessors. Load-Acquire reads, on the other hand, execute before all program-order successors.

ARMv8 introduces a new weaker variant of `dmb`, written `dmb ld`. The `dmb ld` barrier orders 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.

ARM is "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. One

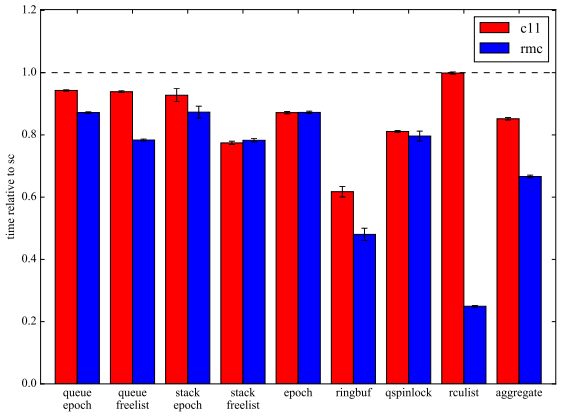


Figure 4. ARMv7 benchmarks

outcome of this is that `dmb ld` can *almost* be used to implement visibility edges from loads to stores: if a load reading from another thread's write is made to execute before a store, Other-multi-copy atomicity ensures that the two writes are observed in the correct order by all processors. But this does not hold if the load reads from a store done by the *same* processor! This snag suggests a workaround: adding a `dmb st` to ensure that any earlier same-processor stores must become visible before subsequent ones. This means that on ARMv8, `dmb ld; dmb st` can serve essentially the same role as Power's `lwsync`: cutting arbitrary visibility edges while being short of a full fence. Perhaps surprisingly, taking advantage of this actually yields performance wins!

6 Evaluation

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 (Figure 4), ARMv8 (Figure 5), and Power (Figure 6). The graphs plot the performance of the C++ low-level atomic and RMC versions relative to that of the SC version. The "aggregate" column shows the geometric mean of the other tests. 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.

The "queue" and "stack" tests are implementations of Michael-Scott concurrent queues [22] and Treiber stacks [27] and each come in two varieties. The "freelist" versions are traditional and use generation counts and Treiber-stack-based free lists for memory management while the "epoch" versions use an elegant memory management technique called epoch-based reclamation [14] that allows threads to register objects for freeing once all threads are done using

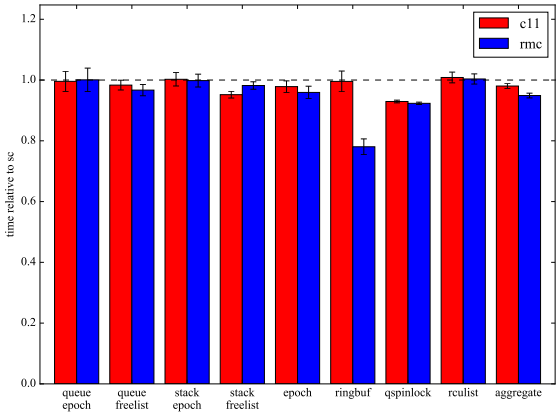


Figure 5. ARMv8 benchmarks

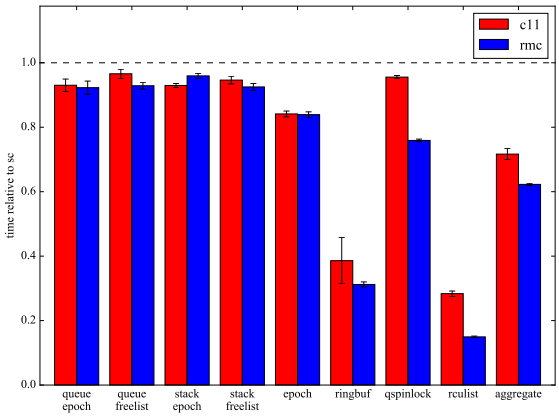


Figure 6. Power benchmarks

them. [2] The "epoch" tests measure the performance of different versions of the epoch library while running stack and queue tests. The "ringbuf" test is a ring-buffer very similar to that discussed in Section . The "qspinlock" test is an implementation of a queue-based spinlock scheme used in the Linux kernel [9].

The "rculist" test measures operations on an RCU-protected linked list. Since RCU is optimized for read-mostly data structures, the test interleaves lookups of list elements and list modifications with one modification every 10000 lookups. Here, list modifications may occur while readers are traversing the list, so reader threads must take care to 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. 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.

Overall, RMC shows a modest performance win on all three tested architectures. On all three, RMC shows modest wins on most (though not all) data structure tests. RMC gets solid wins on the ringbuf test due to its ability ot take advantage of a control dependency to enforce execution order. The epoch library tests show little difference, which makes sense, as C++11 and RMC should generate essentially identical code in the fast path common case. In RCU-protected list manipulation, RMC wins tremendously on ARMv7 and Power by virtue of being able to rely on data dependencies for enforcing execution order. On ARMv8, however, this yields no real speedup over using Load-Acquire instructions! Though not shown in this chart, RMC matches the performance of RCU list search done in the Linux kernel style of attempting to quantify over all possible compiler transformations so as to avoid doing anything that may result in a dependency being optimized away [18]. While this approach is fraught with danger, it has served extremely well from a performance perspective. We consider it a major success that we can match its performance while actually providing a well-defined semantics!

6.1 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. Across our test suit, the (geometric) average slowdown was 1.05x on ARMv7, 1.50x on ARMv8, and 1.65x on POWER— the difference across architectures is due to having more options for implementing constraints on ARMv8 and POWER. The slowdown varies based on the complexity of the test case—RCU list searching on POWER has a slowdown of 1.03x while the very complex qspinlock test has a slowdown of 5.19x.

There are a number of approaches that could be taken to improve compilation time, including attempting to reduce the SMT state space by eliminating unlikely options, adding RMC-tuned heuristics to Z3, and specifying a timeout to Z3 and choosing the best found. We are not particularly troubled by the compilation times, however, as the sort of low-level concurrent code that uses RMC is likely to be a relatively small portion of any real codebase.

7 Related Work

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

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

barriers to recover sequential consistency as part of tools for that purpose. Bender et al. [5] 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.

OpenMP's [6] 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.

There are proposals by McKenney et al. [19, 21] to change the semantics of consume to be more realistically implementable, but nothing seems finalized yet .

Acknowledgments

Joseph Tassarotti provided helpful feedback and input. 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.

References

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't sit on the fence: A static analysis approach to automatic fence insertion. In *CAV*.

[2] ARM Ltd. 2017. *ARM Architecture Reference Manual (ARMv8, for ARMv8-A architecture profile)*. Number ARM DDI 0487B.a.

[3] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Thirty-Ninth ACM Symposium on Principles of Programming Languages*. Philadelphia, Pennsylvania.

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Thirty-Eighth ACM Symposium on Principles of Programming Languages*.

[5] John Bender, Mohsen Lesani, and Jens Palsberg. 2015. Declarative Fence Insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[6] OpenMP Architecture Review Board. 2015. OpenMP Application Programming Interface. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. (Nov. 2015).

[7] Hans-J. Boehm. 2005. Threads Cannot Be Implemented As a Library. In *2005 SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, Illinois.

[8] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness Against TSO. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*.

[9] Jonathan Corbet. 2014. MCS locks and qspinlocks. https://lwn.net/Articles/590243/. (2014).

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

[11] Karl Crary and Michael J. Sullivan. 2015. A Calculus for Relaxed Memory. In *Forty-Second ACM Symposium on Principles of Programming Languages*. Mumbai, India.

[12] Karl Crary, Joseph Tassarotti, and Michael J. Sullivan. 2017. Relaxed Memory Calculus 2.0, draft. https://www.cs.cmu.edu/~jtassaro/papers/rmc2.pdf. (2017).

[13] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS: Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg.

[14] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[15] David Howells and Paul E. McKenney. 2010. Circular Buffers. https://www.kernel.org/doc/Documentation/circular-buffers.txt. (2010).

[16] ISO/IEC 14882:2014. 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).

[17] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (Sept. 1979).

[18] Paul E. McKenney. 2014. Proper Care and Feeding of Return Values From rcu_dereference(). https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.txt. (2014).

[19] Paul E. McKenney, Torvald Riegel, Jeff Preshing, Hans Boehm, Clark Nelson, Oliver Giroux, Lawrence Crowl, JF Bastien, and Michael Wong. 2017. 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).

[20] Paul E. McKenney and John D. Slingwine. 1998. Read-Copy Update: Using Execution History to Solve Concurrency. In *Parallel and Distributed Computing and Systems*.

[21] Paul E. McKenney, Michael Wong, Hans Boehm, Jens Maurer, Jeffrey Yasskin, and JF Bastien. 2017. 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).

[22] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*.

[23] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Thirty-Ninth ACM Symposium on Principles of Programming Languages*.

[24] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER Multiprocessors. In *2011 SIGPLAN Conference on Programming Language Design and Implementation*. San Jose, California.

[25] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM* 53, 7 (July 2010).

[26] Michael J. Sullivan. 2015. rmc-compiler. https://github.com/msullivan/rmc-compiler. (2015).

[27] R. Kent Treiber. 1986. *Systems Programming: Coping with Parallelism*. Technical Report RJ 5118. IBM Almaden Research Center.