Low-level Concurrent Programming Using the Relaxed Memory Calculus

Michael J. Sullivan

Oct 24, 2017

Outline

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 optimizing compilers.

Low-level concurrent programming?

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 optimizing compilers.

- Imperative shared-memory concurrency
- Without locks around all shared data
- “Lock-free algorithms”

- Imperative shared-memory concurrency
- Without locks around all shared data
- “Lock-free algorithms”
- Hard, even under the the best of circumstances
- To be avoided, except when you can’t (perf-critical code, implementation of system libraries, ...)

“The best of circumstances”

- Sequential consistency (SC)
- Threads interleave instructions
- Modifying a single shared memory

Example

```c
int data, flag = 0;

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

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

- Two threads: one wants to send a single message to the other

Example

```
int data, flag = 0;

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

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

- Two threads: one wants to send a single message to the other
- Correctness: recv() only returns the value passed to send()
- If the read from flag returns 1, the read from data must return the sent value

Modernity

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 optimizing compilers.

Modernity
CPU trouble

- No major processor promises SC
- Out-of-order and speculative execution
- Write buffers, caches
- Expensive to maintain SC while having these

Modernity
CPU trouble

- Since most code isn't sharing memory, architectures provide a weaker model
- And explicit fence instructions for when you do share memory

Modernity
CPU trouble

- Can’t view the weak behavior as simply reordering instructions!
- Writes can become visible to CPUs in different orders
- Execution order on the CPU and visibility order inside the memory system are separate phenomena

Modernity
It's not just the CPU

```
int data, flag = 0;

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

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

Compiler might...

Modernity
It's not just the CPU

```
int data, flag = 0;

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

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

Compiler might... hoist the load from flag

```
int data, flag = 0;

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

```
int recv() {
  if (!flag) {
    while (1) continue;
  }
  return data;
}
```

Modernity
It's not just the CPU

```
int data, flag = 0;

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

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

Compiler might... reorder the writes in send()

```
int data, flag = 0;

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

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

Modernity
It's not just the CPU

```
int data, flag = 0;

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

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

Compiler might... reorder the reads in recv()

```
int data, flag = 0;

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

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

Modernity
Compiler trouble

- The loop hoisting is likely
- The reorderings are possible; similar reorderings are more likely
- SC can be violated by: common sub-expression elimination, loop hoisting, some forms of dead code elimination, ...
- Optimizations sound for single-threaded code

Modernity

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 optimizing compilers.

Approaches

- "Threads Cannot be Implemented as a Library"
- This is a language design issue
- Language must define semantics for concurrent programs

Approaches
Starting point

- Provide mutexes that can be used to prevent data races
- (Data race: two threads accessing the same location "at the same time", at least one while writing)
- Promise sequential consistency if no data races
- Most single-threaded optimizations permitted

Approaches
From there?

- And if there <u>are</u> data races?

- What to offer for "low-level" concurrency?

Approaches
From there?

- And if there <ins>are</ins> data races?
    - Java: very weak semantics
    - C++: no semantics ("undefined behavior")
- What to offer for "low-level" concurrency?

Approaches
From there?

- And if there <u>are</u> data races?
  - Java: very weak semantics
  - C++: no semantics ("undefined behavior")
- What to offer for "low-level" concurrency?
  - Wide open design space!

Approaches
Sequentially consistent atomics

- Add locations that may be accessed concurrently
  - Java: `volatile int flag;`
  - C++: `std::atomic<int> flag;`
- “Don’t count” towards data races
- Can use these concurrently without losing SC

Approaches
Sequentially consistent atomics

- Add locations that may be accessed concurrently
  - Java: `volatile int flag;`
  - C++: `std::atomic<int> flag;`
- “Don’t count” towards data races
- Can use these concurrently without losing SC
- Really nice model! But expensive, and strong...

Approaches
C++11 low-level atomics

High-level	Mid-level
Locks	SC atomics

- Want better performance, will accept weaker semantics

Approaches
C++11 low-level atomics

High-level	Mid-level	Low-level
Locks	SC atomics	release, acquire

- Want better performance, will accept weaker semantics
- Mark accesses to atomic locations with “memory orders”: `seq_cst`, `release`, `acquire`, `rel_acq`

Approaches
C++11 low-level atomics

High-level	Mid-level	Low-level	Very low-level
Locks	SC atomics	release, acquire	consume, relaxed, fences

- Want better performance, will accept weaker semantics
- Mark accesses to atomic locations with "memory orders": `seq_cst`, `release`, `acquire`, `rel_acq`
- And `consume` and `relaxed`...
- And memory fences, with memory orders

Approaches
C++11 low-level atomics

- Relations like “synchronizes with” and “happens before” are inferred from these

Approaches
C++11 low-level atomics

- Relations like “synchronizes with” and “happens before” are inferred from these
- “Happens before” isn’t transitive
- “Sequentially consistent” fences can’t restore sequential consistency

Approaches
C++11 low-level atomics

- Relations like "synchronizes with" and "happens before" are inferred from these
- "Happens before" isn't transitive
- "Sequentially consistent" fences can't restore sequential consistency
- We think we can do better

Where RMC fits

C++:

High-level	Mid-level	Low-level	Very low-level
Locks	SC atomics	release, acquire	consume, relaxed, fences

RMC:

High-level	Mid-level
Locks	SC atomics

Where RMC fits

C++:

High-level	Mid-level	Low-level	Very low-level
Locks	SC atomics	release, acquire	consume, relaxed, fences

RMC:

High-level	Mid-level	Low-level
Locks	SC atomics	Ordering constraints

Where RMC fits

C++:

High-level	Mid-level	Low-level	Very low-level
Locks	SC atomics	release, acquire	consume, relaxed, fences

RMC:

High-level	Mid-level	Low-level
Locks	SC atomics	Ordering constraints

- Some subtlety in integrating SC and weaker
- But our main focus is on the low-level

Constraints

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 optimizing compilers.

Constraints

```c
int data, flag = 0;

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

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

- What do we need for this code to work?

Constraints

```
int data, flag = 0;

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

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

- What do we need for this code to work?
- `flag` can be accessed concurrently

Constraints

```
int data, flag = 0;

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

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

- What do we need for this code to work?
- `flag` can be accessed concurrently
- If the write to `flag` is visible to other threads, the write to `data` must be also (vo = visibility order)

Constraints

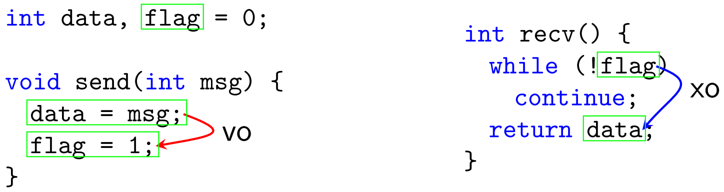
```
int data, flag = 0;

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

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

- What do we need for this code to work?
- `flag` can be accessed concurrently
- If the write to `flag` is visible to other threads, the write to `data` must be also (vo = visibility order)
- The read from `flag` must execute before the read from `data` (xo = execution order)

Constraints



```
int data, flag = 0;

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

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

- What do we need for this code to work?
- `flag` can be accessed concurrently
- If the write to `flag` is visible to other threads, the write to `data` must be also (vo = visibility order)
- The read from `flag` must execute before the read from `data` (xo = execution order)
- The combination ensures that the write to `data` is <u>visible to</u> the read

Constraints

$$\begin{tikzcd}
W[data]=msg \arrow[d, "vo"] & R[flag]=1 \arrow[d, "xo"] \\
W[flag]=1 \arrow[ur, dashed, "rf"] & R[data]=?
\end{tikzcd}$$

- The combination ensures that the write to `data` is visible to the read
- The read must read from it (or a later write)
- (rf = reads from)

Constraints

$$\begin{tikzcd}
W[data]=msg \arrow[d, "vo"] \arrow[dr, dashed, bend left, "rf"] & R[flag]=1 \arrow[d, "xo"] \\
W[flag]=1 \arrow[ur, dashed, bend right, "rf"'] & R[data]=msg
\end{tikzcd}$$

- The combination ensures that the write to `data` is visible to the read
- The read must read from it (or a later write)
- (rf = reads from)

RMC

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 optimizing compilers.

RMC
Concrete syntax

```
int data;
rmc::atomic<int> flag = 0;

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);
}
```

- `L(label, expr)` labels an expression
- `VEDGE` and `XEDGE` establish visibility and execution edges
- `rmc::atomic<int> flag` declares a variable that can be accessed concurrently

Ring buffer

```
typedef struct {
  unsigned char buf[SZ];
  rmc::atomic<unsigned> front, back;
} ring_buf_t;
```

- Example adapted from the Linux Kernel
- Lock-free fixed size FIFO buffer
- One producer, one consumer
- Producer increments `back`, consumer increments `front`.
- Empty when `back - front == 0`, full when `back - front == SZ`.

```c
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front) { // not full
    buf->buf[back % SZ] = c;
    buf->back = back + 1;
  }
}
```

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front) { // not full
    buf->buf[back % SZ] = c;
    buf->back = back + 1;
  }
}

int buf_dequeue(ring_buf *buf) {
  int c = -1;
  unsigned front = buf->front;
  if (front != buf->back) { // not empty
    c = buf->buf[front % SZ];
    buf->front = front + 1;
  }
  return c;
}
```

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front) { // not full
    buf->buf[back % SZ] = c;
    buf->back = back + 1;
  }
}

int buf_dequeue(ring_buf *buf) {
  int c = -1;
  unsigned front = buf->front;
  if (front != buf->back) { // not empty
    c = buf->buf[front % SZ];
    buf->front = front + 1;
  }
  return c;
}
```

- Message passing: values enqueued will be visible to dequeuer

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front) { // not full
    buf->buf[back % SZ] = c;
    buf->back = back + 1;
  }
}

int buf_dequeue(ring_buf *buf) {
  int c = -1;
  unsigned front = buf->front;
  if (front != buf->back) { // not empty
    c = buf->buf[front % SZ];
    buf->front = front + 1;
  }
  return c;
}
```

- Message passing: values enqueued will be visible to dequeuer
- Ensure the value is read before its space is marked as free

```c
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front) { // not full
    buf->buf[back % SZ] = c;
    buf->back = back + 1;
  }
}

int buf_dequeue(ring_buf *buf) {
  int c = -1;
  unsigned front = buf->front;
  if (front != buf->back) { // not empty
    c = buf->buf[front % SZ];
    buf->front = front + 1;
  }
  return c;
}
```

- Message passing: values enqueued will be visible to dequeuer
- Ensure the value is read before its space is marked as free
- Don't write a value until we know its space is free

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  XEDGE(echeck, insert);
  VEDGE(insert, eupdate);
  unsigned back = buf->back;
  if (back != SZ + L(echeck, buf->front)) {
    L(insert, buf->buf[back % SZ] = c);
    L(eupdate, buf->back = back + 1);
  }
}
```

Pushes
Rationale

- Consider the following (broken!) code, which could be a snippet from a mutual exclusion algorithm

```
lock1 = 1;
if (!lock2) {
  // Critical section
}
```

```
lock2 = 1;
if (!lock1) {
  // Critical section
}
```

Pushes
Rationale

- Consider the following (broken!) code, which could be a snippet from a mutual exclusion algorithm

```
lock1 = 1;
if (!lock2) {
  // Critical section
}
```

```
lock2 = 1;
if (!lock1) {
  // Critical section
}
```

- Could let both threads into critical section
- Can't fix this with visibility or execution edges

- Consider the following (broken!) code, which could be a snippet from a mutual exclusion algorithm

```
lock1 = 1;                      lock2 = 1;
if (!lock2) {                   if (!lock1) {
  // Critical section             // Critical section
}                               }
```

- Could let both threads into critical section
- Can't fix this with visibility or execution edges
- Need write to become globally visible before read

Pushes

- Pushes are globally visible actions–visible before everything that executes after them
- Visibility between pushes is thus a total order
- Doesn't do much on its own; combined with execution and visibility edges to constrain behavior
- Architecturally, a full fence. Stalls execution until things are visible

Pushes
Using pushes

$$\begin{tikzcd}
\text{W[lock1]} = 1 \arrow[d, "\text{vo}"] & \text{W[lock2]=1} \arrow[d, "\text{vo}"] \\
\text{push} \arrow[d, "\text{xo}"] & \text{push} \arrow[d, "\text{xo}"] \\
\text{R[lock2]=?} & \text{R[lock1]=?}
\end{tikzcd}$$

- Push is visibility after the write, execution before the read

Pushes
Using pushes

$$\begin{tikzcd}
W[lock1] = 1 \arrow[d, "vo"] & W[lock2]=1 \arrow[d, "vo"] \\
push \arrow[r, dashed, "vo"'] \arrow[d, "xo"] & push \arrow[d, "xo"] \\
R[lock2]=? & R[lock1]=?
\end{tikzcd}$$

- Push is visibility after the write, execution before the read
- One of the pushes needs to be visible to the other

Pushes
Using pushes

$$\begin{tikzcd}
W[lock1] = 1 \arrow[d, "vo"] \arrow[drr, dashed, bend left, "rf"] & W[lock2]=1 \arrow[d, "vo"] \\
push \arrow[r, dashed, "vo"'] \arrow[d, "xo"] & push \arrow[d, "xo"] \\
R[lock2]=? & R[lock1]=1
\end{tikzcd}$$

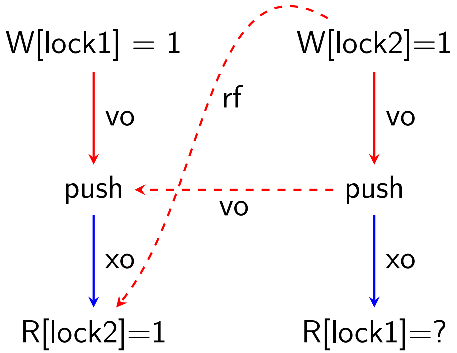
- Push is visibility after the write, execution before the read
- One of the pushes needs to be visible to the other
- Which makes the write visible to the other thread’s read

Pushes
Using pushes

$$\begin{tikzcd}
W[lock1] = 1 \arrow[d, "vo"] & W[lock2]=1 \arrow[d, "vo"] \\
push \arrow[d, "xo"] & push \arrow[l, dashed, "vo"] \arrow[d, "xo"] \\
R[lock2]=? & R[lock1]=?
\end{tikzcd}$$

- Push is visibility after the write, execution before the read
- One of the pushes needs to be visible to the other
- Which makes the write visible to the other thread's read

Pushes
Using pushes



- Push is visibility after the write, execution before the read
- One of the pushes needs to be visible to the other
- Which makes the write visible to the other thread's read

Pushes
Push syntax

```
VEDGE(write, flush);
XEDGE(flush, read);
L(write, lock1 = 1);
L(flush, rmc::push());
if (!L(read, lock2)) {
  // Critical section
}
```

Pushes
Push syntax - push edges

```
PEDGE(write, read);
L(write, lock1 = 1);
if (!L(read, lock2)) {
  // Critical section
}
```

That’s the heart of it

- Two core concepts: execution and visibility order
- Pushes, which are defined in terms of them

That’s the heart of it

- Two core concepts: execution and visibility order
- Pushes, which are defined in terms of them
- More to talk about, but all about how to specify constraints!

Advanced constraint specification
Spinlocks

```
void spinlock_lock(spinlock_t *lock) {
  while (test_and_set(&lock->locked) == 1)
    continue;
}

void spinlock_unlock(spinlock_t *lock) {
  lock->locked = 0;
}
```

- What constraints do we need to use these in a program?

Advanced constraint specification
Spinlocks

```
void spinlock_lock(spinlock_t *lock) {
  while (test_and_set(&lock->locked) == 1)
    continue;
}

void spinlock_unlock(spinlock_t *lock) {
  lock->locked = 0;
}
```

- What constraints do we need to use these in a program?
- The body of the critical section must execute after `spinlock_lock` and before `spinlock_unlock`
- And visible before `spinlock_unlock`

Advanced constraint specification
Spinlocks

```
void spinlock_lock(spinlock_t *lock) {
  while (test_and_set(&lock->locked) == 1)
    continue;
}

void spinlock_unlock(spinlock_t *lock) {
  lock->locked = 0;
}
```

- What constraints do we need to use these in a program?
- The body of the critical section must execute after `spinlock_lock` and before `spinlock_unlock`
- And visible before `spinlock_unlock`
- Only allowing constraints from label-to-label is cumbersome and anti-modular

Advanced constraint specification
Pre and post edges

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

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

- We add special `pre` and `post` labels to allow specifying edges to all program order predecessors and successors

Advanced constraint specification
Pre and post edges

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

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

- We add special `pre` and `post` labels to allow specifying edges to <u>all</u> program order predecessors and successors
- `pre` and `post` labels are very important for interface boundaries

Advanced constraint specification
Cross-loop edges

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

$$\begin{tikzcd}
\text{W[x]=0} \arrow[r] \arrow[r, bend right, "\text{vo}"'] \arrow[rrr, bend right, "\text{vo}"'] & \text{W[y]=10} \arrow[r] & \text{W[x]} = 1 \arrow[r] \arrow[r, bend right, "\text{vo}"'] & \text{W[y]} = 11
\end{tikzcd}$$

- By default, edges established with all subsequent actions
- Including those in later function invocations

Advanced constraint specification
Bound edges

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

$$\begin{tikzcd}
\mathrm{W[x]}=0 \arrow[r] \arrow[r, bend right, "\mathrm{vo}"'] & \mathrm{W[y]}=10 \arrow[r] & \mathrm{W[x]} = 1 \arrow[r] \arrow[r, bend right, "\mathrm{vo}"'] & \mathrm{W[y]} = 11
\end{tikzcd}$$

- Edge is “bound” in the region dominated by the edge declaration in the CFG

Advanced constraint specification
Cross-function constraints

- Also have a way to specify fine-grained cross-function edges
- Important for RCU-style workloads using data dependencies
- Won't go into details

Theory
Overview

- Formalized typed core-calculus
- Dynamic semantics explicitly accounts for out-of-order and speculative execution
- Very weak, to future-proof against new hardware

Theorems

- Progress and Preservation
- Interleaving actions with pushes gives sequential consistency
- Data-race-free executions are sequentially consistent
- All formalized in Coq

Practical?

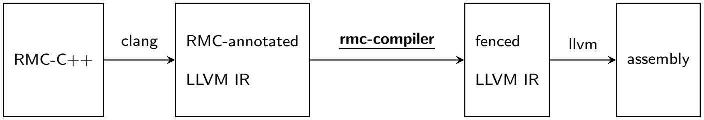
Explicit programmer-specified constraints on execution order and visibility of writes are a **practical** approach for low-level concurrent programming in the presence of modern hardware and optimizing compilers.

Practical?

- Generates efficient code
- Usable model to program with and reason about
- Rigorously specified

Implementation

- Have a working compiler based on LLVM
- Targets x86, ARMv7, ARMv8, POWER
- `https://github.com/msullivan/rmc-compiler`



Implementation
How to compile?

- Compilation driven by the constraints
- Insert code to ensure the constraints hold

Implementation
Compiling visibility

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

  L(wflag, flag = 1);
}
```

- Visibility edge means `wdata` must be visible to anything that sees `wflag`
- ARM does this with the `dmb` instruction

Implementation
Compiling visibility

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

  L(wflag, flag = 1);
}
```

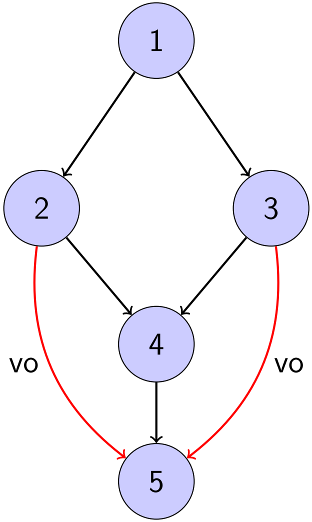
$\longrightarrow$

```
void send(int msg) {

  data = msg;
  dmb();
  flag = 1;
}
```

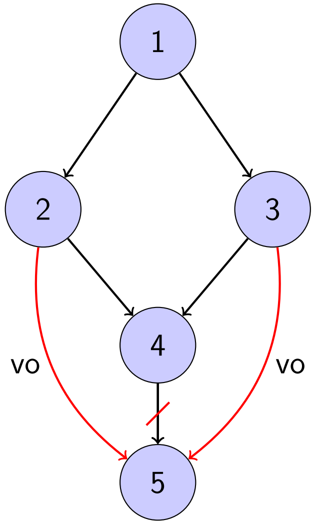
- Visibility edge means `wdata` must be visible to anything that sees `wflag`
- ARM does this with the `dmb` instruction

Implementation
Compiling visibility



- Insert fences to cut all paths
- Minimize cost of fences
- We use an SMT solver to do it

Implementation
Compiling visibility



- Insert fences to cut all paths
- Minimize cost of fences
- We use an SMT solver to do it

Implementation
Compiling execution

```
XEDGE(rflag, rdata);
while (!L(rflag, flag))
  continue;

return L(rdata, data);
```

- Execution edge means `rflag` must execute before `rdata`
- On ARM: lots of ways to force things to execute in order!

Implementation
Compiling execution

```
XEDGE(rflag, rdata);
while (!L(rflag, flag))
  continue;

return L(rdata, data);
```

$\longrightarrow$

```
while (!flag)
  continue;
dmb();
return data;
```

- Execution edge means `rflag` must execute before `rdata`
- On ARM: lots of ways to force things to execute in order!
- dmb still works

Implementation
Compiling execution

```
XEDGE(rflag, rdata);
while (!L(rflag, flag))
  continue;

return L(rdata, data);
```

$\longrightarrow$

```
while (!flag)
  continue;
isb();
return data;
```

- Execution edge means `rflag` must execute before `rdata`
- On ARM: lots of ways to force things to execute in order!
- Or a control dependency and then an `isb`

Implementation
Compiling execution

```
XEDGE(rflag, rdata);
while (!L(rflag, flag))
  continue;

return L(rdata, data);
```

$\longrightarrow$

```
while (!flag)
  continue;
dmb_ld();
return data;
```

- Execution edge means `rflag` must execute before `rdata`
- On ARM: lots of ways to force things to execute in order!
- On ARMv8, `dmb ld` is a general execution edge

Implementation
Compiling execution

```
XEDGE_HERE(rptr, rdata);
foo *p = L(rptr, ptr);
return L(rdata, p->data);
```

- Data dependencies ensure ordering
- Being able to take advantage of this is a big selling point

Implementation
Compiling execution

```
XEDGE_HERE(rptr, rdata);
foo *p = L(rptr, ptr);
return L(rdata, p->data);
```

$\longrightarrow$

```
foo *p = ptr;
return p->data;
```

- Data dependencies ensure ordering
- Being able to take advantage of this is a big selling point

Implementation
Release/Acquire

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

- ARMv8 basically built the C++11 model into hardware!
- “Store-Release” and “Load-Acquire” (but really SC)

Implementation
Release/Acquire

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

$\longrightarrow$

```
void send(int msg) {

  data = msg;
  flag.store_release(1);
}
```

- ARMv8 basically built the C++11 model into hardware!
- “Store-Release” and “Load-Acquire” (but really SC)

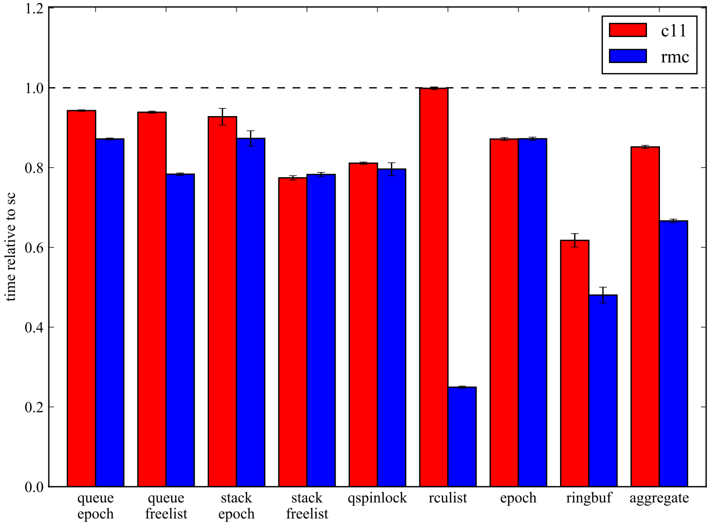
- “Generates efficient code”
- “Usable model to program with and reason about”
- Evaluated this with case studies
- Implemented RMC, C++11, SC versions

- Concurrency primitives: mutexes, rwlocks, sequence locks
- Lock-free data structures: queues, stacks, ring buffers
- “read-copy-update” (RCU) and client code

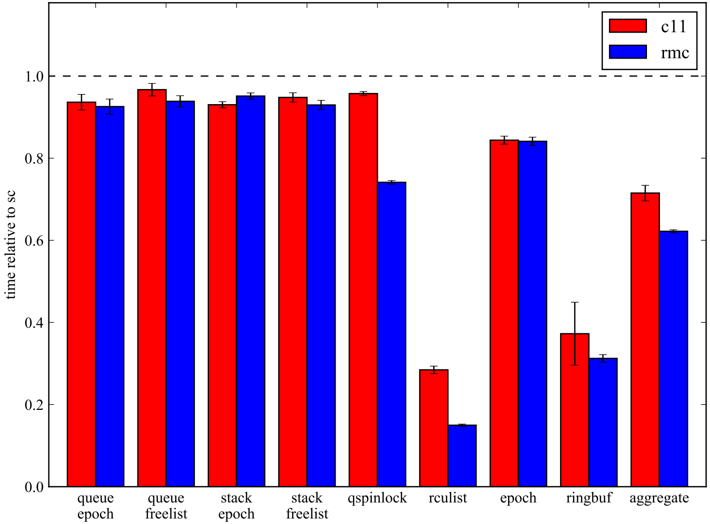
Efficient?

- Used our case studies for benchmarking
- ARMv7: NVIDIA Jetson TK1 4 CPU ARM Cortex-A15
- ARMv8: ODROID-C2 4 CPU ARM Cortex-A53
- POWER 8 from IBM Power Systems Academic Cloud

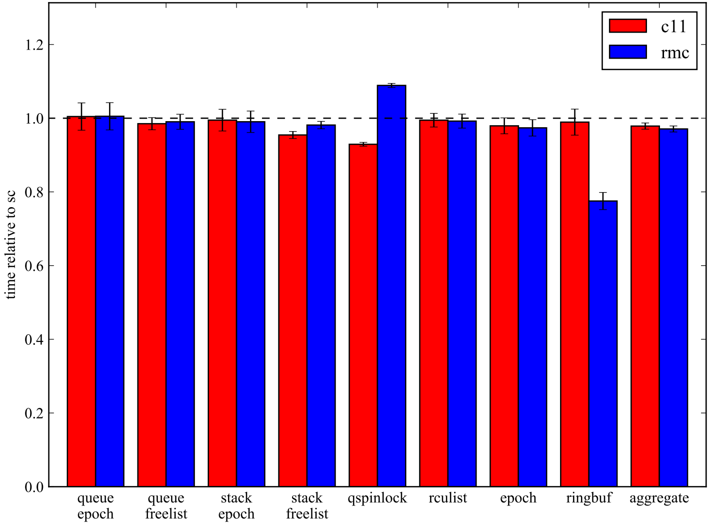
Efficient?
ARMv7



Efficient?
POWER



Efficient?
ARMv8



Usable?
Impressions

- I found the RMC versions easier to understand, reason about, and modify
- Smallest benefit was for simple cases (like stacks)
- Difference was more pronounced for complex constraints
- Especially when using fences

Usable?
Impressions

- I found the RMC versions easier to understand, reason about, and modify
- Smallest benefit was for simple cases (like stacks)
- Difference was more pronounced for complex constraints
- Especially when using fences
- I found the easiest way to use C++11 fences was to essentially run the RMC compiler in my head

Usable?
My favorite one

- Hans Boehm: "Can Seqlocks Get Along With Programming Language Memory Models?"
- In C++11, yes, but "exceptionally unnatural" – needs an "acquire" fence for an unlock operation
- They get along with RMC just fine!

Usable?
“User Study” ($N = 1$)

- Two undergrads implemented lock-free data structures with RMC and C11 for 15-418 final project
- RMC versions performed better
- “using [RMC] was significantly more straightforward than manually using the C11 atomic intrinsics.”

Usable?
Takeaways

- Relative ordering is <u>the</u> fundamental concept
- Writing low-level concurrent code requires careful thought about relative ordering

Usable?
Takeaways

- Relative ordering is <u>the</u> fundamental concept
- Writing low-level concurrent code requires careful thought about relative ordering
- So make it explicit!

Usable?
Takeaways

- Modifying traditional low-level concurrent code requires reconstructing the necessary ordering invariants!

Usable?
Takeaways

- Modifying traditional low-level concurrent code requires reconstructing the necessary ordering invariants!
- From algorithm's design, comments, tea leaves...

Usable?
Takeaways

- Modifying traditional low-level concurrent code requires reconstructing the necessary ordering invariants!
- From algorithm's design, comments, tea leaves...
- So make it explicit!

Conclusion

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 optimizing compilers.

- The compiler is up on github if you want to try it!

Thank you!

Carnegie Mellon University

EMARHAVIL HEAVY INDUSTRIES

What about `volatile`?

What about `volatile`?

- What about `volatile`?

What about `volatile`?

- What about `volatile`?
- By spec, `volatile` is totally orthogonal
- “Access to volatile objects are evaluated strictly according to the rules of the abstract machine” (N4296 §1.9.8.1)
- Races on `volatile` locations still undefined behavior

What about `volatile`?

- What about `volatile`?
- By spec, `volatile` is totally orthogonal
- “Access to volatile objects are evaluated strictly according to the rules of the abstract machine” (N4296 §1.9.8.1)
- Races on `volatile` locations still undefined behavior
- In practice, `volatile` can be used to tame compiler behavior (Linux does this)

  ```
  #define ACCESS_ONCE(x) (*(volatile __typeof__(x) *)&(x))
  ```

- Doesn’t say anything about hardware, though, so still need to handle that

Bonus!
C++11 - SC fragment

```
int data
std::atomic<int> flag = 0;

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

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

Bonus!
C++11 - release/acquire

```
int data
std::atomic<int> flag = 0;

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

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

Bonus!
C++11 - release/acquire fences

```
int data
std::atomic<int> flag = 0;

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

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

Bonus!
C++11 in RMC

```
int load_acquire(rmc::atomic<int> *ptr) {
    XEDGE(load, post);
    return L(load, *ptr);
}

void store_release(rmc::atomic<int> *ptr, int val) {
    VEDGE(pre, store);
    L(store, *ptr = val);
}
```

Bonus!
C++11 in RMC - generalized it

```
template <typename T>
T load_acquire(rmc::atomic<T> *ptr) {
    XEDGE(load, post);
    return L(load, *ptr);
}

template <typename T>
void store_release(rmc::atomic<T> *ptr, T val) {
    VEDGE(pre, store);
    L(store, *ptr = val);
}
```

More comparison to C++11

- We give the compiler more flexibility in how to implement things
- C++11 ring buffers would do two releases, two acquires
- We can get a lot of the benefit of consume without the large complexities involved

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front.load(std::memory_order_acquire)) { // not full
    buf->buf[back % SZ] = c;
    buf->back.store(back + 1, std::memory_order_release);
  }
}
```

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front.load(std::memory_order_acquire)) { // not full
    buf->buf[back % SZ] = c;
    buf->back.store(back + 1, std::memory_order_release);
  }
}

int buf_dequeue(ring_buf *buf) {
  int c = -1;
  unsigned front = buf->front;
  if (front != buf->back.load(std::memory_order_acquire)) { // not empty
    c = buf->buf[front % SZ];
    buf->front.store(front + 1, std::memory_order_release);
  }
  return c;
}
```

Bonus!
Cross-loop edges

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

$$\begin{tikzcd}
\mathrm{W[x]}=0 \arrow[r] \arrow[r, bend right, "\mathrm{vo}"'] \arrow[rrr, bend right, "\mathrm{vo}"'] & \mathrm{W[y]}=10 \arrow[r] & \mathrm{W[x]} = 1 \arrow[r] \arrow[r, bend right, "\mathrm{vo}"'] & \mathrm{W[y]} = 11
\end{tikzcd}$$

Bonus!
Cross-loop edges

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

$$\begin{tikzcd}
W[x]=0 \arrow[r] & W[y]=10 \arrow[r] \arrow[r, bend right, "vo"'] & W[x] = 1 \arrow[r] & W[y] = 11
\end{tikzcd}$$

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
\text{W[data]=msg} \arrow[r, dashed, "\text{rf}"] & \text{R[data]=msg} \arrow[d, "\text{dmb}"] & \text{R[flag]=1} \arrow[d, "\text{addr}"] \\
 & \text{W[flag]=1} \arrow[ur, dashed, "\text{rf}"] & \text{R[data]=?}
\end{tikzcd}$$

- Weak memory can not simply be viewed as reordering of instructions!
- The address dependency from the read to the write has meaning (it constrains the coherence order of flag), but doesn't mean msg needs to be visible everywhere

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
\text{W[data]=msg} \arrow[r, dashed, "\text{rf}"] \arrow[rrd, dashed, bend left, "\text{rf}"] & \text{R[data]=msg} \arrow[d, "\text{dmb}"] & \text{R[flag]=1} \arrow[d, "\text{addr}"] \\
& \text{W[flag]=1} \arrow[ru, dashed, "\text{rf}"] & \text{R[data]=msg}
\end{tikzcd}$$

- Weak memory can <u>not</u> simply be viewed as reordering of instructions!
- The address dependency from the read to the write has meaning (it constrains the coherence order of flag), but doesn't mean msg needs to be visible everywhere

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
\text{W[data]=msg} \arrow[r, dashed, "\text{rf}"] & \text{R[data]=msg} \arrow[d, "\text{addr}"] & \text{R[flag]=1} \arrow[d, "\text{addr}"] \\
& \text{W[flag]=1} \arrow[ur, dashed, "\text{rf}"] & \text{R[data]=?}
\end{tikzcd}$$

- Weak memory can not simply be viewed as reordering of instructions!
- The address dependency from the read to the write has meaning (it constrains the coherence order of flag), but doesn't mean msg needs to be visible everywhere

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
\mathrm{W[data]{=}msg} \arrow[r, dashed, "\mathrm{rf}"] & \mathrm{R[data]{=}msg} \arrow[d, "\mathrm{addr}"] & \mathrm{R[flag]{=}1} \arrow[d, "\mathrm{addr}"] \\
 & \mathrm{W[flag]{=}1} \arrow[ur, dashed, "\mathrm{rf}"] & \mathrm{R[data]{=}0}
\end{tikzcd}$$

- Weak memory can <u>not</u> simply be viewed as reordering of instructions!
- The address dependency from the read to the write has meaning (it constrains the coherence order of flag), but doesn't mean msg needs to be visible everywhere

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
W[data]=msg \arrow[r, dashed, "rf"] & R[data]=msg \arrow[d, "vo"] & R[flag]=1 \arrow[d, "xo"] \\
 & W[flag]=1 \arrow[ur, dashed, "rf"'] & R[data]=?
\end{tikzcd}$$

- Visibility pushes out visibility to other threads transitively
- Execution doesn't

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
\text{W[data]=msg} \arrow[r, dashed, "\text{rf}"] \arrow[drr, dashed, bend left, "\text{rf}"] & \text{R[data]=msg} \arrow[d, "\text{vo}"] & \text{R[flag]=1} \arrow[d, "\text{xo}"] \\
 & \text{W[flag]=1} \arrow[ur, dashed, "\text{rf}"] & \text{R[data]=msg}
\end{tikzcd}$$

- Visibility pushes out visibility to other threads transitively
- Execution doesn't

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
\text{W[data]=msg} \arrow[r, dashed, "\text{rf}"] & \text{R[data]=msg} \arrow[d, "\text{xo}"] & \text{R[flag]=1} \arrow[d, "\text{xo}"] \\
& \text{W[flag]=1} \arrow[ur, dashed, "\text{rf}"] & \text{R[data]=?}
\end{tikzcd}$$

- Visibility pushes out visibility to other threads transitively
- Execution doesn't

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
W[data]=msg \arrow[r, dashed, red, "rf"] & R[data]=msg \arrow[d, blue, "xo"] & R[flag]=1 \arrow[d, blue, "xo"] \\
 & W[flag]=1 \arrow[ur, dashed, red, "rf"] & R[data]=0
\end{tikzcd}$$

- Visibility pushes out visibility to other threads transitively
- Execution doesn't

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
R[x]=? \arrow[d, "xo"] & R[y]=1 \arrow[d, "xo"] \\
W[y]=1 \arrow[ur, dashed, "rf"] & W[x]=1
\end{tikzcd}$$

- There is a notion of the order things “execute in” - trace order
- Execution order, visibility order, reads from all must agree with it

Bonus!
Execution vs. visibility

$$\begin{tikzcd}
R[x]=? \arrow[d, "xo"] & R[y]=1 \arrow[d, "xo"] \\
W[y]=1 \arrow[ur, dashed, bend right, "rf"'] & W[x]=1 \arrow[ul, dashed, "rf"]
\end{tikzcd}$$

- There is a notion of the order things “execute in” - trace order
- Execution order, visibility order, reads from all must agree with it

```
void buf_enqueue(ring_buf *buf, unsigned char c) {
  unsigned back = buf->back;
  if (back != SZ + buf->front) { // not full
    buf->buf[back % SZ] = c;
    buf->back = back + 1;
  }
}

int buf_dequeue(ring_buf *buf) {
  int c = -1;
  unsigned front = buf->front;
  if (front != buf->back) { // not empty
    c = buf->buf[front % SZ];
    buf->front = front + 1;
  }
  return c;
}
```

$$\begin{tikzcd}
\text{echeck} \arrow[d, "\text{xo}"'] & \text{dcheck} \arrow[d, "\text{xo}"'] \\
\text{write} \arrow[d, "\text{vo}"'] \arrow[r, dashed, "\text{rf}"] & \text{read} \arrow[d, "\text{xo}"'] \\
\text{eupdate} & \text{dupdate} \arrow[uul, dashed, "\text{rf}"']
\end{tikzcd}$$

Dead store elimination

$$
\begin{array}{l}\texttt{x = 1;}\\ \texttt{y = 1;}\\ \texttt{x = 2;}\end{array}
\longrightarrow
\begin{array}{l}\texttt{y = 1;}\\ \texttt{x = 2;}\end{array}
$$

Another thread could observe `y = 1` without observing `x = 1`

Bonus!
Non-atomics

- Atomic locations too strong to use for everything
- Each read reads from exactly one write
- All writes eventually reach all threads
- Followed C++ in having non-atomics with undefined behavior for races

Bonus!
Sequential consistency

```
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);
}
```

- "SC atomics" could be implemented using pushes
- Stronger than we need, though
- Extra strength has real performance costs
- Added an efficient SC fragment

Advanced constraint specification
Cross-function constraints

```
foo *get_foo() {
  return L(rptr, ptr);
}

int stuff() {

  foo *p = get_foo();
  return L(rdata, p->data);
}
```

- How to specify cross-function edges?

Advanced constraint specification
Cross-function constraints

```
foo *get_foo() {
  return L(rptr, ptr);
}

int stuff() {
  XEDGE(getfoo, rdata);
  foo *p = L(getfoo, get_foo());
  return L(rdata, p->data);
}
```

- How to specify cross-function edges?

Advanced constraint specification
Cross-function constraints

```
foo *get_foo() {
  return L(rptr, ptr);
}

int stuff() {
  XEDGE(pre, rdata);
  foo *p = get_foo();
  return L(rdata, p->data);
}
```

- How to specify cross-function edges?

Advanced constraint specification
Cross-function constraints

```
foo *get_foo() {
  return L(rptr, ptr);
}

int stuff() {
  XEDGE(pre, rdata);
  foo *p = get_foo();
  return L(rdata, p->data);
}
```

- How to specify cross-function edges?
- These solutions rule out using data dependencies

Advanced constraint specification
Cross-function constraints

```
foo *get_foo() {
  XEDGE_HERE(rptr, ret);
  foo *p = L(rptr, ptr);
  return LGIVE(ret, p);
}

int stuff() {
  XEDGE_HERE(get, rdata);
  foo *p = LTAKE(get, get_foo());
  return L(rdata, p->data);
}
```

- How to specify cross-function edges?
- These solutions rule out using data dependencies
- `LGIVE` and `LTAKE` allow returning values to be treated as a pseudo-action

Related Work

- Java memory model (Manson et al. 2005)
- C++ memory model (Boehm and Adve 2008, Batty et al. 2010)
- Sarkar, et al. 2011; POWER operational model
  - Direct inspiration for our system
- Alglave et al. 2014; generic framework, "leapfrogging writes"
- Jagadeesan et al. 2010; operational model for Java
  - Our mechanism for speculation adapted from this
- Boehm and Demsky 2014; "out-of-thin-air" results worse than we realized