A Calculus for Relaxed Memory

Karl Crary and Michael J. Sullivan

Carnegie Mellon University
POPL ’15, Mumbai

January 17, 2015
Relaxed Memory Calculus

- A new approach to language memory models for concurrency
  - That is, specifying what writes are available to reads
  - In the presence of optimizing compilers and SMP machines
- Based around specifying visibility and execution orderings
- Suitable for use with C/C++
- With a mechanized metatheory
• Concurrent programming is hard, even under the best of circumstances

• Sequential consistency: threads interleave instructions, modifying a single shared memory
Concurrency?

- Concurrent programming is hard, even under the best of circumstances
- Sequential consistency: threads interleave instructions, modifying a single shared memory
- Languages designed so that if locks are used to rule out data races, events are sequentially consistent
• Concurrent programming is hard, even under the best of circumstances

• Sequential consistency: threads interleave instructions, modifying a single shared memory

• Languages designed so that if locks are used to rule out data races, events are sequentially consistent

• But sometimes that isn’t good enough (perf-critical code, implementation of system libraries, ...)

Concurrency?
Concurrent Message Passing

```c
int data, flag;

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.
- Nope!
int data, flag;

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
int data, flag;

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
• Nope!
Concurrency?
Message passing: What goes wrong

int data, flag;

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

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

• Compiler could reorder writes in send, hoist the load out of the loop, ...
• CPU has out of order and speculative execution, multilevel caches, ...
void send(int msg) {
    data = msg;
    flag = 1;
}

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

• What do we need for this code to work?
int data, flag;

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

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

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

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

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

- What do we need for this code to work?
- 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)
int data, flag;

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

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

• What do we need for this code to work?
• 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 visible to the read
RMC
Constraints

- 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)
The combination ensures that the write to `data` is *visible* to the read

- The read must read from it (or a later write)
- \((rf = \text{reads from})\)
**RMC**

*Key concepts*

- Have the programmer explicitly specify these constraints
- Allow specification of visibility and execution ordering
int data, 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);
}

• \texttt{L(label, expr)} labels an expression
• \texttt{VEDGE} and \texttt{XEDGE} establish visibility and execution edges
The C++11 memory model marks accesses to atomic memory locations with various “memory orders”. Relations like “synchronizes with” and “happens before” are inferred from these. “Happens before” isn’t transitive.
• Nicer to specify the key relations \textit{directly}
• And it gives the compiler more flexibility
typedef struct {
    unsigned char buf[BUF_SIZE];
    unsigned front, back;
} ring_buf_t;

#define ring_inc(v) (((v) + 1) % BUF_SIZE)

- Example adapted from the Linux Kernel
- Lock-free fixed size FIFO buffer
- One producer, one consumer
- Producer modifies back, consumer modifies front.
- Empty when back == front, full when ring_inc(back) == front.
void buf_enqueue(ring_buf_t *buf, unsigned char c) {
    unsigned back = buf->back;
    if (ring_inc(back) != buf->front) { // not full
        buf->buf[back] = c;
        buf->back = ring_inc(back);
    }
}

int buf_dequeue(ring_buf_t *buf) {
    int c = -1;
    unsigned front = buf->front;
    if (front != buf->back) { // not empty
        c = buf->buf[front];
        buf->front = ring_inc(front);
    }
    return c;
}
void buf_enqueue(ring_buf_t *buf, unsigned char c) {
    unsigned back = buf->back;
    if (ring_inc(back) != buf->front) { // not full
        buf->buf[back] = c;
        buf->back = ring_inc(back);
    }
}

int buf_dequeue(ring_buf_t *buf) {
    int c = -1;
    unsigned front = buf->front;
    if (front != buf->back) { // not empty
        c = buf->buf[front];
        buf->front = ring_inc(front);
    }
    return c;
}
void buf_enqueue(ring_buf_t *buf, unsigned char c) {
    unsigned back = buf->back;
    if (ring_inc(back) != buf->front) { // not full
        buf->buf[back] = c;
        buf->back = ring_inc(back);
    }
}

int buf_dequeue(ring_buf_t *buf) {
    int c = -1;
    unsigned front = buf->front;
    if (front != buf->back) { // not empty
        c = buf->buf[front];
        buf->front = ring_inc(front);
    }
    return c;
}

• Message passing: values enqueued will be visible to dequeuer
```c
void buf_enqueue(ring_buf_t *buf, unsigned char c) {
    unsigned back = buf->back;
    if (ring_inc(back) != buf->front) { // not full
        buf->buf[back] = c;
        buf->back = ring_inc(back);
    }
}

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

- Message passing: values enqueued will be visible to dequeuer
- Ensure the value is read before its space is marked as free
void buf_enqueue(ring_buf_t *buf, unsigned char c) {
    unsigned back = buf->back;
    if (ring_inc(back) != buf->front) { // not full
        buf->buf[back] = c;
        buf->back = ring_inc(back);
    }
}

int buf_dequeue(ring_buf_t *buf) {
    int c = -1;
    unsigned front = buf->front;
    if (front != buf->back) { // not empty
        c = buf->buf[front];
        buf->front = ring_inc(front);
    }
    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_t *buf, unsigned char c) {
    XEDGE(echeck, insert);
    VEDGE(insert, eupdate);
    unsigned back = buf->back;
    if (ring_inc(back) != L(echeck, buf->front)) {
        L(insert, buf->buf[back] = c);
        L(eupdate, buf->back = ring_inc(back));
    }
}

int buf_dequeue(ring_buf_t *buf) {
    XEDGE(dcheck, read);
    XEDGE(read, dupdate);
    int c = -1;
    unsigned front = buf->front;
    if (front != L(dcheck, buf->back)) {
        c = L(read, buf->buf[front]);
        L(dupdate, buf->front = ring_inc(front));
    }
    return c;
}
- Consider the following (broken!) code, which could be a snippet from a mutual exclusion algorithm

```c
lock1 = 1;
if (!lock2) {
    // Critical section
}
lock2 = 1;
if (!lock1) {
    // Critical section
}
```
Consider the following (broken!) code, which could be a snippet from a mutual exclusion algorithm:

```c
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
Pushes

- Pushes are globally visible actions
- Totally ordered
- Doesn’t do much on its own; combined with execution and visibility edges to constrain behavior
Pushes
Using pushes

\[ W[\text{lock1}] = 1 \quad \text{and} \quad W[\text{lock2}] = 1 \]

\[ \begin{align*}
\text{vo} \\
\text{push} \\
\text{xo} \\
R[\text{lock2}] = ? \\
\end{align*} \]

\[ \begin{align*}
\text{vo} \\
\text{push} \\
\text{xo} \\
R[\text{lock1}] = ? \\
\end{align*} \]

- Push is visibility after the write, execution before the read
Pushes
Using pushes

\[ W[\text{lock1}] = 1 \quad W[\text{lock2}] = 1 \]

\[ \text{vo} \quad \text{vo} \]

\[ \text{push} \quad \text{push} \]

\[ \text{xo} \quad \text{xo} \]

\[ R[\text{lock2}] = ? \quad R[\text{lock1}] = ? \]

- Push is visibility after the write, execution before the read
- One of the pushes needs to be visible to the other
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
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(write1, push1);
XEDGE(push1, read1);
L(write1, lock1 = 1);
L(push1, PUSH);
if (!L(read1, lock2)) {
// Critical section
}

VEDGE(write2, push2);
XEDGE(push2, read2);
L(write2, lock2 = 1);
L(push2, PUSH);
if (!L(read2, lock1)) {
// Critical section
}
Theory
Overview

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

Coherence order

- Coherence order - order on writes to each location
- Key technical device
- Ensures single threaded computation works as expected
• Progress and Preservation
• Interleaving actions with pushes gives sequential consistency
• Race free executions are sequentially consistent
• Progress and Preservation
• Interleaving actions with pushes gives sequential consistency
• Race free executions are sequentially consistent
• All formalized in Coq
Implementation

- Compiler needs to preserve execution order
- On x86, visibility and execution order come for free
- On ARM, visibility order can be enforced with a fence (\texttt{dmb}); execution order allows more options
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
Conclusion

- RMC is a new memory model built around explicitly specifying visibility and execution orderings
- Details about the formalism and model are in the paper
- Implementation is being developed on top of Clang/LLVM
Thank you!
```c
int load_acquire(int *ptr) {
    XEDGE(load, post);
    return L(load, *ptr);
}

void store_release(int *ptr, int 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