Memory Ordering

Joseph Tassarotti
Why is concurrency hard?

Writing concurrent programs is hard, even with sequentially consistent memory:

1. Hard to reason about – have to think about all possible thread interleavings
2. Hard to pick the right building blocks – locks? atomic instructions? semaphores? condition variables?
3. Hard to debug – tricky to reproduce buggy interleavings
Writing concurrent programs is hard, even with sequentially consistent memory:

1. Hard to reason about — have to think about all possible thread interleavings
2. Hard to pick the right building blocks — locks? atomic instructions? semaphores? condition variables?
3. Hard to debug — tricky to reproduce buggy interleavings

It gets worse!
Weak memory

- Intuitively, we often think about concurrent programs in terms of interleavings of threads.
- Weak memory models permit executions that do not correspond to interleavings. For example with release-acquire semantics:

\[
[x]_{rel} := 1 \quad \parallel \quad [y]_{rel} := 1 \quad \parallel \quad r_1 = [x]_{acq} \quad \parallel \quad r_3 = [y]_{acq} \\
r_2 = [y]_{acq} \quad \parallel \quad r_4 = [x]_{acq}
\]

- After running this, it’s possible for \(r_1 = 1\), \(r_2 = 0\), \(r_3 = 1\), and \(r_4 = 0\)!
Why is concurrency harder with weak memory?

Writing concurrent programs is harder with weak memory:

1. **Harder** to reason about – have to think about all possible thread interleavings ... and executions that do not correspond to interleavings!

2. **Harder** to pick the right building blocks – locks? atomic instructions? semaphores? condition variables? ... now add different consistency level options to them (e.g. C++11 has 4 different types of memory stores)

3. **Harder** to debug – tricky to reproduce buggy interleavings ... and now there are yet more options, some of which can only occur on certain hardware!
Why is concurrency harder with weak memory?

Writing concurrent programs is **harder** with weak memory:

1. **Harder** to reason about – have to think about all possible thread interleavings . . . and executions that do not correspond to interleavings!

2. **Harder** to pick the right building blocks – locks? atomic instructions? semaphores? condition variables? . . . now add different consistency level options to them (e.g. C++11 has 4 different types of memory stores)

3. **Harder** to debug – tricky to reproduce buggy interleavings . . . and now there are yet more options, some of which can only occur on certain hardware!

This is, in fact, worse!
How do these papers help?


→ Makes execution deterministic so same ordering for every run

*Efficient processor support for DRFx, a memory model with exceptions* by Abhayendra Singh, Daniel Marino, Satish Narayanasamy, Todd Millstein, and Madan Musuvathi.

→ Raise exception if non-sequentially consistent behavior occurs
RCDC - High Level

**RCDC:** a relaxed consistency deterministic computer by Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman.

- Making execution deterministic makes it easier to reproduce bugs
- Earlier work by authors/others had shown how to determinize sequentially consist and total store order memory models.
  - But it’s slow to do so.
- Paper shows even weaker “data race free” model can be determinized more efficiently than total store order.
What’s the Data-Race-Free (DRF) Model?

“Simplified” model for non-expert programmers:

- Divide variables into *synchronization* variables (e.g. status of a lock) and *data* variables (everything else).
- Use special, expensive, synchronized primitives to manipulate synchronization variables (e.g. for locking/unlocking)
  - Compiler promises not to re-order things past these, and emits fences to prevent hardware from doing so
- If no data races on data variables, you get sequentially consistent behavior. If there are races, it’s either undefined (C/C++11) or too complicated to understand (Java).
- This is weaker than what the hardware is actually giving you, but it’s simple to understand.
Abstract model:

- Each processor has a FIFO “store buffer”
- Periodically, these buffers are flushed to memory
- A processor reads from its own store buffer before looking at memory
- Fence instructions force a buffer to flush.
- This is what’s found on x86 (more or less).
- It gives more guarantees than DRF (no completely undefined behavior).
DMP-TSO

(Figure 1, Devietti et. al.)
(Figure 1, Devietti et al.)
Commit Phase

(Figure 4, Devietti et. al.)
How does synchronization (locking) work?

- No round robin sequential synchronization phase, so how to determinize?
- When thread wants to lock:
  1. Block until we’re the thread that has executed fewest instructions!
  2. Do a CAS on lock status variable.
  3. If we succeed, we now have lock. Check whether the last lock was (1) done in a different quantum, or (2) held by the same thread. If so, no need for fence. Otherwise we need a fence, so end quantum.
- Why no need for fence?
  - If it’s the same thread locking, then no synchronization necessary.
  - If it’s in a different quantum, we already synchronized during commit phase.
Results

(Figure 8, Devietti et. al.)
Determinization is sensitive to cache size. **Different cache sizes cause different executions.** So what do we do with user submitted bug reports if they have different hardware?

Strange anomalous behavior due to weak consistency can still occur – still hard to track down, or even notice in some cases.

What is performance for lock free algorithms? Did not implement support for, but claim it’s possible.

Lots of other sources of non-determinism in concurrent programs: user interaction, network traffic, hardware, other processes on same machine.
Questions/Critique

- Other paper (DRFx) addresses this somewhat by triggering exception and halting execution if data race occurs that results in non-SC behavior.
- But... that makes it inappropriate for kernel. Actually, Linux kernel explicitly makes use of non-SC behavior rather frequently for performance reasons.
- Recall: why did we move to non-SC behavior in the first place? Performance.
- DRFx adds overhead and terminates program if non-SC behavior happens. Less overhead than SC hardware seems to require, but still changes cost-benefit ratio – is the extra complexity worth it?