EECS 570
Lecture 15
Relaxed Consistency

Winter 2019

Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Special Thanks: Daniel Lustig, NVIDIA, for RISC-V memory model slides
Special Thanks: Don Porter, UNC, for RCU slides
Announcements

• Programming Assignment 2
  - Waypoint due today
  - Submit via Canvas

• No lecture next Monday - ISPASS

• Project Milestone 2
  - Slides due 3/27
  - Meetings 3/29
  - Prepare a brief slide deck in lieu of a written report
  - Submit via Canvas
Readings

For today:

- Alglave et al - Frightening Small Children and Disconcerting Adults: Concurrency in the Linux Kernel. ASPLOS 2018

For Wednesday 3/15:

- Gharachorloo et al - Two Techniques to Enhance the performance of Memory Consistency Models - ICPP 1991
Relaxed Consistency
A WIDE RANGE OF MEMORY MODELS

Sequential Consistency

Low Performance

There is a big cliff here called “multi-copy-atomicity”

IBM Power

NVIDIA GPUs

Hard for Programmers

Total Store Ordering (TSO)

RISC-V (RVWMO)

Hard for Implementers

Note: diagram obviously not to scale, just a rough picture 😊
MULTI-COPY ATOMICITY

A load may only return a value from:

- An earlier store from the same hart ("hardware thread")
- A store that is globally visible

In other words, a store may not “peek” into a neighbor hart’s private store buffer.
OPERATIONAL VS. AXIOMATIC

In modern practice, at ISA level, two common modeling approaches:

**Axiomatic:** define a set of criteria ("axioms") to be satisfied
- Executions permitted unless they fail one or more axioms

**Operational:** define a golden abstract machine model
- Executions forbidden unless producible when executing this model

Ideally: figure out how to meet in the middle (can be difficult!)
- lots of gray area, obscure code, etc.
SEQUENTIAL CONSISTENCY [LAMPORT ‘79]

Axiomatic

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order

3. Loads return the value written by the latest store to the same address in the total order

Operational

1. Harts take turn executing instructions. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Loads return the value written by the most recent preceding store to the same address
SEQUENTIAL CONSISTENCY [LAMPORT ‘79]

**Axiomatic**

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order

3. Loads return the value written by the latest store to the same address in the total order

**Operational**

1. Harts take turn executing instructions. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Loads return the value written by the most recent preceding store to the same address

**Preserved Program Order (PPO)**

**Load Value Axiom**

**Global memory order**
Basic Relations in Axiomatic models

[Alglave]

Abstract executions \((E, po, addr, data, ctrl, rmw)\) contain:

- \(E\), the set of events (e.g., reads, writes);
- \(po\), the program order, specifies instruction order in a thread after evaluating conditionals and unrolling loops;
- \(addr\), \(data\), and \(ctrl\) are the address, data, and control dependency relations in \(po\), always starting from a read.
- \(rmw\) links the read of a read-modify-write to its write.

Execution witnesses \((rf, co)\) contain:

- the reads-from relation \(rf\), which determines where reads take their value from. For each read \(r\) there is a unique write \(w\) to the same location s.t. \(r\) takes its value from \(w\).
- the coherence order relation \(co\), representing the history of writes to each location. It is a total order over writes to the same location, starting with the initialising write.
Selected Derived LK Relations [Alglave]

- The from-reads relation consists of one step of reads-from backwards, then one step of coherence: \( fr := rf^{-1}; co; \)
- The communication relation gathers reads-from, coherence and from-reads: \( com := rf \cup co \cup fr; \)
- The dependency relation gathers address and data (but not control) dependencies: \( dep := addr \cup data; \)

... ...

- The preserved program order relation, which captures cases where local program order is preserved in memory order due to intra-thread dependence or fence
  \( ppo := rrdep*; (to-r \cup to-w \cup fence) \)
- The happens-before relation, which combines ppo, reads across threads (rfe), and transitive ordering from local fences
  \( hb := ((prop \setminus id) \cap int) \cup ppo \cup rfe \)
- The propagates-before relation, which augments happens-before with orderings arising from strong fences
  \( pb := prop ; strong-fence ; hb* \)
What’s the point? Finding cycles

• If there is a cycle in a set of relations, then the corresponding sequence of events is provably impossible, as it results in a contradiction

• Linux Kernel model from Alglave:
  - $acyclic(po\text{-}loc \cup com)$ (Scpv)
  - $empty(rmw \cap (fre ; coe))$ (At)
  - $acyclic(hb)$ (Hb)
  - $acyclic(pb)$ (Pb)
Proving Disallowed Executions Can’t Happen

Litmus Test

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i1) [x] ← 1</td>
<td>(i3) r1 ← [y]</td>
</tr>
<tr>
<td>(i2) [y] ← 1</td>
<td>(i4) r2 ← [x]</td>
</tr>
<tr>
<td>Under SC: Forbid r1=1, r2=0</td>
<td></td>
</tr>
</tbody>
</table>

- Construct all possible executions of litmus tests using μArch axioms
- Ensure disallowed behaviors lead to μArch happens-before cycles → unobservable
TOTAL STORE ORDERING (SPARC, X86, RVTSO)

Axiomatic

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects program order, except Store→Load ordering

3. Loads return the value written by the latest store to the same address in program or memory order (whichever is later)

Operational

1. Harts take turn executing steps. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Stores execute in two steps: 1) enter store buffer, 2) drain to memory

4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address
TOTAL STORE ORDERING (SPARC, X86, RVTSO)

Axiomatic

\[
ppo := (\text{program order}) - W \rightarrow R
\]

\[
\text{acyclic}(ppo \cup \text{rfe} \cup \text{co} \cup \text{fr} \cup \text{fence})
\]

\[
\text{acyclic}(po\_loc \cup \text{rf} \cup \text{co} \cup \text{fr})
\]

Operational

[Alglave et al., TOPLAS '09]

[Seewell et al., CACM '10]
Sun's definitions… Dependence Order

- A refinement of program order (<p)

- Dependence order (<d) captures the minimal subset of (<p) that guarantees self-consistent execution traces. \( X <p Y \) implies \( X <d Y \) if at least one of the following is true:
  - The execution of \( Y \) is conditional on \( X \) and \( S(Y) \) (\( Y \) is a store)
  - \( Y \) reads a register that is written by \( X \)
  - \( X \) and \( Y \) access the same memory location and \( S(X) \) and \( L(Y) \)

- Dependence order captures what an out-of-order core is allowed to do (ignoring exceptions)
Sun’s “Relaxed Memory Order” (RMO)

• Formal requirements [v9 architecture manual]:
  
  - $X <d Y \land L(X) \Rightarrow X <M Y$
    - RMO will maintain dependence order if preceding insn. is a load
  
  - $M(X, Y) \Rightarrow X <M Y$
    - MEMBAR instructions order memory operations
  
  - $Xa <p Ya \land S(Y) \Rightarrow X <M Y$
    - Stores to the same address are performed in program order
Execution in RMO w/ fence

- "Fence" indicates ordering affects correctness
  - Retirement stalls at Fence
  - Typically, accesses after fence don’t issue

```
CPU pipeline

<table>
<thead>
<tr>
<th></th>
<th>St A</th>
<th>Fence</th>
<th>Ld B</th>
<th>St C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>retired</td>
<td>can’t issue</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>St A</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ld B</td>
<td>Idle</td>
<td></td>
</tr>
<tr>
<td>St C</td>
<td>Idle</td>
<td></td>
</tr>
</tbody>
</table>
```
RMO: Programmer’s Perspective

• Programmer must specify MEMBARS wherever they are needed
  □ This is hard; Specifying minimum barriers is harder
  □ E.g., lock and unlock (from v9 ref. manual; w/o branch delays)

LockWithLDSTUB (lock)
  retry:  
  ldastub [lock],%10 
  tst %10 
  be out 
  loop:  
  ldaub [lock],%10 
  tst %10 
  bne loop 
  ba,a retry 
  out:  
  membar #LoadLoad | #LoadStore

UnLockWithLDSTUB (lock)
  membar #StoreStore  !RMO and PSO only 
  membar #LoadStore    !RMO only 
  stub %g0,[lock]
RMO: Compiler’s Perspective

• Sweet, sweet freedom!
  - Compiler may freely re-order between fences
  - Also across fences if the fence allows it (partial fences)
  - Note: still can’t reorder stores to same address
  - Programmer’s problem if the fences are wrong...
RMO: HW Perspective

- Unordered, coalescing post-retirement store buffer
  - Just like PSO

- Out-of-order execution!
  - Need a standard uniprocessor store queue
  - Fence instruction implementation
    - Easy – stall at issue
    - Hard – retire to store buffer and track precise constraints
Weak Ordering (WO)

- Loads and stores can be labeled as “sync”
  - No reordering allowed across sync instructions
Release Consistency (RC)

- Specialize loads and stores as “acquires” and “releases”
  - Load at top of critical section is an acquire
  - Store at bottom of critical section is a release
  - Allows reordering into critical sections, but not out

![Diagram showing release consistency and weak consistency](Diagram)

Release Consistency (RCpc)

u cannot perform with respect to any other processor until u is performed
RISC-V WEAK MEMORY ORDERING (RVWMO)

Axiomatic

1. There is a total order on all memory operations. The order is non-deterministic.

2. That total order respects thirteen specific patterns (next slide)

3. Loads return the value written by the latest store to the same address in program or memory order (whichever is later)

Operational

1. Harts take turn executing steps. The order is non-deterministic.

2. Each hart executes its own instructions in order

3. Multiple steps for each instruction (see spec Appendix B)

4. Loads first try to forward from the store buffer. If that fails, they return the value written by the most recent preceding store to the same address
RVWMO

Axiomatic (App. B.2)

ppo := (13 rules, on next slide)

acyclic(pfo U rfe U co U fr)

acyclic(pfo_loc U rf U co U fr)

Operational (App. B.3)

This is an alternative presentation of the RVWMO memory model in operational style. It aims to admit exactly the same extensional behaviour as the axiomatic presentation: for any given program, admitting an execution if and only if the axiomatic presentation allows it.

The axiomatic presentation is defined as a predicate on complete candidate executions. In contrast, this operational presentation has an abstract microarchitectural flavor: it is expressed as a state machine, with states that are an abstract representation of hardware machine states, and with explicit out-of-order and speculative execution (but abstracting from more implementation-specific microarchitectural details such as register renaming, store buffers, cache hierarchies, cache protocols, etc.). As such, it can provide useful intuition. It can also construct executions incrementally, making it possible to interactively and randomly explore the behavior of larger examples, while the axiomatic model requires complete candidate executions over which the axioms can be checked.

The operational presentation covers mixed-size execution, with potentially overlapping memory accesses of different power-of-two byte sizes. Misaligned accesses are broken up into single-byte accesses.

An interactive version of the model, together with a library of litmus tests, is provided online: http://www.cl.cam.ac.uk/~pes20/rwmo. This is integrated with a fragment of the BIBS-V ISA semantics (RVW and A) expressed explicitly in Naiill (https://github.com/naimo-project/naai).

Below is an informal introduction of the model states and transitions. The description of the formal model starts in the next subsection.

**Terminology:** In contrast to the axiomatic presentation, here every memory operation is either a load or a store. Hence, AMOs give rise to two distinct memory operations, a load and a store. When used in conjunction with “instruction”, the terms “load” and “store” refer to instructions that give rise to such memory operations. As such, both include AMO instructions. The term “memory” refers to shared memory that is accessible to all processors. The BIBS-V or release consistency. The memory model.

**Hart states:**

![Hart 0 ... Hart n](image)

**Shared Memory**
RVWMO PPO RULES IN A NUTSHELL

- **Preserved Program Order**: if A appears before B in program order, and A and B match one of the patterns below, then A appears before B in global memory order.

![Diagram showing RVWMO PPO rules: Preserved Program Order conditions with examples of operations like Store, Load, AMO/SC, etc.](image-url)
**WO/RC: Programmer Perspective**

- A new way of thinking: **programmer-centric memory models**
  - If you annotate syncs correctly, your program will behave like SC
  - E.g., Data-Race-Free-0 (DRF0)
    - Accesses are either normal or sync
    - Sync accesses are sequentially consistent, and order normal accesses
    - Data races among normal accesses are prohibited
    - **DRF0 programs appear as if they ran under SC**
  - Lot’s of ongoing work on race detection
RCU in a nutshell

- Think about data structures that are mostly read, occasionally written
  - Like the Linux dcache
- RW locks allow concurrent reads
  - Still require an atomic decrement of a lock counter
  - Atomic ops are expensive
- Idea: Only require locks for writers; carefully update data structure so readers see consistent views of data
Motivation
(from Paul McKenney’s Thesis)

Performance of RW lock only marginally better than mutex lock
Principle (1/2)

- Locks have an acquire and release cost
  - Substantial, since atomic ops are expensive
- For short critical regions, this cost dominates performance
Principle (2/2)

- Reader/writer locks may allow critical regions to execute in parallel.
- But they still serialize the increment and decrement of the read count with atomic instructions.
  - Atomic instructions performance decreases as more CPUs try to do them at the same time.
- The read lock itself becomes a scalability bottleneck, even if the data it protects is read 99% of the time.
RCU: Split the difference

- One of the hardest parts of lock-free algorithms is concurrent changes to pointers
  - So just use locks and make writers go one-at-a-time
  - But, make writers be a bit careful so readers see a consistent view of the data structures
- If 99% of accesses are readers, avoid performance-killing read lock in the common case
Example: Linked lists

This implementation needs a lock

Reader goes to B

B’s next pointer is uninitialized; Reader gets a page fault
Example: Linked lists

Reader goes to C or B---either is ok

Garbage collect C after all readers finished
Garbage collection

- Part of what makes this safe is that we don’t immediately free node C
  - A reader could be looking at this node
  - If we free/overwrite the node, the reader tries to follow the ‘next’ pointer
  - Uh-oh
- How do we know when all readers are finished using it?
  - Hint: No new readers can access this node: it is now unreachable
Quiescence

- Trick: Linux doesn’t allow a process to sleep while traversing an RCU-protected data structure
  - Includes kernel preemption, I/O waiting, etc.
- Idea: If every CPU has called schedule() (quiesced), then it is safe to free the node
  - Each CPU counts the number of times it has called schedule()
  - Put a to-be-freed item on a list of pending frees
  - Record timestamp on each CPU
  - Once each CPU has called schedule, do the free