EECS 570
Lecture 16
Relaxed Consistency
Winter 2020
Prof. Satish Narayanasamy

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

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Singh, Smith, Torrellas and Wenisch. Special Thanks: Daniel Lustig, NVIDIA, for RISC-V memory model slides Special Thanks: Don Porter, UNC, for RCU slides
Relaxed Consistency
A WIDE RANGE OF MEMORY MODELS

Note: diagram obviously not to scale, just a rough picture 😊

There is a big cliff here called “multi-copy-atomicity”
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. Global memory order

2. Preserved Program Order (PPO)

3. Load Value Axiom
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

- 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:
  - $\text{acyclic}(\text{po-loc} \cup \text{com})$ (Scpv)
  - $\text{empty}(\text{rmw} \cap (\text{fre} ; \text{coe}))$ (At)
  - $\text{acyclic}(\text{hb})$ (Hb)
  - $\text{acyclic}(\text{pb})$ (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] \leftarrow 1)</td>
<td>(i3) (r1 \leftarrow [y])</td>
</tr>
<tr>
<td>(i2) ([y] \leftarrow 1)</td>
<td>(i4) (r2 \leftarrow [x])</td>
</tr>
</tbody>
</table>

Under SC: Forbid \(r1=1, r2=0\)

- Construct all possible executions of litmus tests using \(\mu\)Arch axioms
- Ensure disallowed behaviors lead to \(\mu\)Arch 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

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

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

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

Operational

---

Fig. 8. The message passing pattern mp with lightweight fence and ppo (forbidden)

[Alglave et al., TOPLAS '09]

[Sewell 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
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:  ldstub [lock],%10
            tst %10
            be out
    loop:   ldu [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 (RC) and weak consistency (WCsc)]
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(ppo U rfe U co U fr)

acyclic(po_loc U rf U co U fr)

Operational (App. B.3)

B.3 An Operational Memory Model

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 flavour: 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 execution incrementally, making it possible to interactively and randomly explore the behaviour 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. Marigold 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.cc.gatech.edu/~jpark/emu. This is integrated with a fragment of the RISC-V ISA semantics (RISC-V and X) expressed explicitly in Sisal (https://github.com/sisal-project/sisal).

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, AXOs 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 AXO instructions. The term "acquire" refers to an AXO instruction with a memory reference that doesn’t occur in the same instruction. The term "release" refers to an AXO instruction with a memory reference that occurs in the same instruction. All AXO instructions are RCOPe or RCPe.

Figure A.13: A variant of the LB litmus test (outcome forbidden)
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.

- **Overlap:** Store 
- **Load:** 
- **AMO/SC:** Overlap 
- **Fence:** B
- **.aq:** with pr/pw/sr/sw set appropriately
- **.rl:** RCsc
- **LR:** except ctrl deps. where B is a load
- **Addr/ctrl/data dep.:** “(addr|data);rfi” or “addr;po”
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
Our Approach: InvisiFence

- Key departure: apply to weakly-ordered system
  - Straightforward hardware; fewest stalls to address
- Augment with familiar deep speculation mechanisms
  - Violation detection: read/write bits in cache
  - Version management: clean to L2 before 1\textsuperscript{st} write
- Result: eliminate fence stalls (up to 13% speedup)
  - No fine-grained (per-store) tracking
  - Fast & simple commit and rollback
  - Conventional memory system
- For strong ordering: speculate more ("implicit fences")
  - Bonus: can even eliminate LSQ snooping! (a la \cite{Ceze07})
Roadmap

- InvisiFence for weak ordering
- Generalizing InvisiFence to stronger models
- Subsuming in-window speculation
- Conclusions
Background: Weak Ordering

- Relaxes ordering except at programmer-inserted fences
  - Allows unordered store buffer to hide store misses
- Unordered, coalescing store buffers $\rightarrow$ simple, scalable
  - Cache-like organization
  - Store hits skip store buffer; only one entry per miss
  - Result: largely eliminate capacity stalls of FIFO store buffers

- However, still incur consistency-induced stalls
  - ...even with in-window speculation (LSQ snooping)
  - Fences: drain store buffer (stall until empty)
  - Atomic ops: stall until has write permission
InvisiFence For Weak Ordering

- Add deep speculation to eliminate stalling on fences

- Mechanism: register ckpt + 2 bits per L1 cache line
  - Similar HW to other deep speculation (TLS, TM, Cherry...)

- Initiate speculation at fence instructions
  - Detect violations via cache coherence protocol
  - Preserve non-speculative data in L2 (facilitates rollback)

- Speculation ends when store buffer becomes empty
  - Commit by flash-clearing read/write bits
InvisiFence Hardware

Baseline:
- OoO pipeline
- LSQ snooping
- Writeback L1 & L2
- Invalidation-based CC
- Coalescing store buffer

InvisiFence extensions:
- Register checkpoint
- 2 bits per L1 cache line
- 2 bits per SB entry
InvisiFence: Example

Fence wants to retire...
Initiate speculation
Speculatively retire fence
...but store miss outstanding

Key:
- S: Store
- F: Fence
- L: Load
- Other Insn

Diagram:
- P0: Tail ROB Head
- L1: r/w?
- SB: r/w?
- Dirty
- L2

Diagram elements:
- Ckpt
- Store
- Load
- Other Insn
InvisiFence: Violation Detection

At store retirement: Set write bit

Key
- Store
- Fence
- Load
- OtherInsn
InvisiFence: Violation Detection

At load retirement:
Set read bit
InvisiFence: Violation Detection

To detect violations:

snoop bits

Key:
- S: Store
- F: Fence
- L: Load
- Other Insn

Diagram:
- P0: Tail,ROB, Head, Ckpt
- P1: L1, r/w, SB, r/w
- L2

Dirty

Dirty

Dirty

Dirty
InvisiFence: Version Management

P0

Tail ROB Head

S

Ckpt

L1 r/w?

SB r/w?

Dirty S

Dirty

R

Dirty

L2

Dirty

Clean to L2 before 1st speculative write

P1

Key

S Store

F Fence

L Load

Other Insn
InvisiFence: Version Management

P0

Tail  ROB  Head

L1  r/w?

SB  r/w?

Key

Dirty  w

Dirty  r

Dirty  w

[Diagram of InvisiFence with states and operations]

P1
InvisiFence: Version Management

Can always recover non-spec version from L2 (no custom storage)
InvisiFence: Rollback

P0

Tail    ROB    Head

P1

L1  r/w?

SB  r/w?

L2

Dirty  Dirty  Dirty

Key

Store  Load  Fence  Other Insn
InvisiFence: Rollback

P0

Tail
ROB
Head

Ckpt

P1

L1 r/w?

SB r/w?

Dirty w
w

Dirty w
w

Flash-clear spec. dirty blocks

Flash-clear bits
Flash-inval spec. dirty blocks

L2

Dirty

Key

S Store
F Fence
L Load
Other Insn
InvisiFence: Rollback

L0

Tail ROB Head

F

P1

Begin re-execution

Rollback: Fast & simple

L1 r/w? SB r/w?

L2 Dirty

Key

Store Load Fence Other Insn
InvisiFence: When to Commit?

Back to speculation: Store returns

Key:
- **S**: Store
- **F**: Fence
- **L**: Load
- **Other Insn**: Other instructions
InvisiFence: When to Commit?

P0

Tail ROB Head

L1 r/w?

SB r/w?

P1

Move store & r/w bit from SB to L1

Key

Store Load
Fence Other Insn

Dirty

Dirty

Dirty

Dirty

S

F

Other Insn

r/w?

r/w?
InvisiFence: When to Commit?

No outstanding stores: Legal to commit
InvisiFence: Commit

P0

Tail | ROB | Head

L1  | r/w? | SB  | r/w?

L2

Dirty | w
Dirty | b
Dirty | r
Dirty | w

Discard checkpoint

Flash-clear bits

Key

S: Store
F: Fence
L: Load
Other Insn
InvisiFence: Commit

P0

L1 r/w? SB r/w?

L2

Dirty

Commit: Fast & simple

Key

S Store
F Fence
L Load
Other Insn
InvisiFence Performance

SimFlex simulation of 16-node directory-based SPARC MP
SPARC’s RMO (similar to Alpha, ARM, PowerPC)
InvisiFence Performance

InvisiFence eliminates fence stalls without violations

But what about models requiring stronger ordering?
Generalizing InvisiFence for Strong Ordering

• Strong models impose additional ordering constraints
  - Processor Consistency (x86, TSO): ordering between stores
  - Sequential Consistency: ordering between all operations

• These constraints are conceptually “implicit fences”
  - e.g., for SC: every operation is “implicit fence”

• InvisiFence can handle these just like explicit fences!
  - Increases speculation frequency...

No other hardware changes
Strong Ordering Performance (SC)

Violations are negligible (3% slowdown from IF-RMO)

How does this compare to prior work?
Strong Ordering Performance (SC)

Comparison to Atomic Sequence Ordering [Wenisch `07]:
Both eliminate stalls
Roadmap

• InvisiFence for weak ordering
• Generalizing InvisiFence to stronger models
• Subsuming in-window speculation
• Conclusions
Key Idea: Continuous Speculation

[Hammond'04, Ceze'07]

- Prior work: subsume LSQ snooping via continuous spec.
  - Execution divided into continuous speculative chunks
  - Deep spec. tracks loads from execution to chunk commit
  - Commit a chunk once all stores complete & all loads retire

- Existing designs acquire store permissions at commit
  - Lazy conflict detection (lowers vulnerability to violations)
  - Shown to be useful for other applications (TM, debugging, ...)
  - Requires extensions to conventional memory systems

- InvisiFence can also support continuous speculation
  - Eliminates LSQ snooping with local commit
  - Like prior work, pipelines commit with second checkpoint
Continuous Speculation Performance

To reduce rollbacks: “Commit on Violation”

- Temporarily defer conflicting requests
Continuous Speculation Performance

Normalized Runtime

- Violation
- SB drain
- SB full
- Other

apache
zeus
oracle oltp-db2
dss-db2
barnes
ocean
IF-cont+ (with commit on violation) achieves IF-sc performance without LSQ snooping.
Conclusions

InvisiFence eliminates stalls from weak ordering
• Without per-store buffering
• With fast & simple commit and abort
• Using a conventional memory system

Same hardware can provide strong ordering
• Adjust policy to start speculation
• InvisiFence-SC: within 3% of InvisiFence-RMO

Subsume in-window speculation mechanisms
• Add continuous speculation + commit on violation
• InvisiFence-SC performance without LSQ snooping