EECS 570
Lecture 14
Programming Language
MCMs and Spec. MCM Implementations

Winter 2022
Prof. Yatin Manerkar

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

Announcements

- PA1 grades have been released
  - Mean: 90.12
  - Median: 90.12
  - St dev: 5.81

- PA2 Waypoint due this Friday

- Final Project Milestone meetings next week
  - Signup link will be posted on Canvas/Piazza

- We know about the space issues on bane; DCO is investigating
MCMs: Specification vs Implementation

• For a given ISA, there is only one specification of a particular MCM for it
  - e.g., for x86, there is only one specification of TSO

• However, there are multiple possible implementations of an MCM for a given ISA
  - e.g., for x86-TSO, you can implement it by having an SC processor
  - You can have an in-order processor with a store buffer
  - You can have a store buffer and read your own write early
  - You can speculatively reorder memory operations (will see later today)
  - ...

Relaxing all Order

- Motivation: Out-of-order execution & multiple load misses

Now Ld E can complete even though earlier insn aren’t done
Two ISA Approaches to enforce order

• Approach 1: Using explicit “fence” (aka memory barrier)
  - Sun’s Relaxed Memory Order (RMO), Alpha, PowerPC, ARM
    
    Ld, St, ...
    
    L L S S
    Fence ↓↓↓↓ Enforces order if bit is set
    L S L S
    Ld, St, ...

• Approach 2: Annotate loads/stores that do synchronization
  - Weak Ordering, Release Consistency (RC)
  - Data-Race-Free-0 (DRF0) – prog. language-level model

    Load.acquire     Lock1
    ...
    Store.release    Lock1
More 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
  - Assuming $Y$ is a load to memory location $a$,
    \[
    \text{Value}(La) = \text{Value}(\text{Max}_{<m} \{ S \mid Sa <M La \text{ or } Sa <p La \} )
    \]
    - where $\text{Max}_{<M}\{..\}$ selects the most recent element with respect to the memory order and where $\text{Value()}$ yields the value of a particular memory transaction.
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
    - See VSync [Oberhauser et al. ASPLOS 2021]
  - Below: lock and unlock (from v9 ref. manual; w/o branch delays)
  - RMO also does not preserve same address ld-ld ordering!

```
LockWithLDSStub(lock)
    retry:  ldstub [lock],%10
            tst %10
            be out
    loop:   ldub [lock],%10
            tst %10
            bne loop
            ba,a retry
    out:    membar #LoadLoad | #LoadStore

UnLockWithLDSStub(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)

- [Dubois et al. ISCA 1986]

- 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 (“roach motel”)

![Diagram of release consistency](image)
Release Consistency (RC)

- Two flavours: RCsc and RCpc
  - In RCsc, special accesses are sequentially consistent with respect to each other
  - In RCpc, special accesses are processor consistent with respect to each other
- The difference has significant ramifications!
  - In RCpc, even adding enough synchronization to eliminate data races does not make the program behave like SC!
  - In RCpc, ordering is not maintained between releases and subsequent acquires
    - Why do this?
  - In RCpc, releases do not become visible to all cores at the same time
    - Why do this?
Data Races

A program has a **data race** if it has an execution in which two accesses to the **same address** on **different threads** where **at least one is a write** and **at least one is not a synchronization access** are not ordered by synchronization accesses.

Note: The definition of races can vary depending on the context.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: x = 1;</td>
<td>C: r1 = y;</td>
</tr>
<tr>
<td>B: y = 1;</td>
<td>D: z = 1;</td>
</tr>
<tr>
<td>E: r2 = x;</td>
<td></td>
</tr>
</tbody>
</table>
WO/RC: Programmer Perspective

- A new way of thinking: programmer-centric memory models
  - If you annotate syncs correctly, your program will behave like SC
    - WHY?
  - E.g., Data-Race-Free-0 (DRF0) [Adve and Hill ISCA 1990]
    - 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
      - Similar idea in [Gharachorloo et al. ISCA 1990]: “Properly-[Labelled]” Programs
  - SC-for-DRF forms the basis for programming language memory models
The Ramifications of SC-for-DRF

- Hardware can freely reorder instructions in between acquire and release operations
  - No one else can tell!
- Compilers can freely reorder code in between acquire and release operations
  - Again, no one else can tell!
- So does this solve all memory consistency problems?
  - Not quite!
    - The hardware and compiler still need to maintain ISA-level MCM and PL MCM guarantees
    - Programmers still need to ensure their programs are race-free
    - What about cases where we want some atomics that are not sequentially consistent? (for performance reasons)
    - Next: programming language MCMs
Programming Language MCMs
Programming Language MCMs

• What should a programming language MCM be?
  • SC?
    □ Too restrictive for e.g., compiler
    □ Also unnecessary!
  • Key idea: Get programmer to annotate accesses as data or synchronization
    □ Compiler (and indirectly the hardware) then knows when it can reorder/optimize and when it can’t
    □ One end of spectrum: HW that assumes DRF, e.g., DeNovo
  • Key idea 2: Make programmer ensure that program is DRF
    □ Advantage: will behave as if it was SC
    □ Assigning semantics to racy programs is hard
• Outstanding issue: out-of-thin-air values (will revisit later)
The C++ Memory Model

- Include lock/unlock in the language so people can use it to synchronize their code

- Include atomic variables that people can use for more fine-grained synchronization
  - Akin to synchronization accesses in RC/DRF
  - Easiest to understand: sequentially consistent atomics
  - Also acquire, release, and relaxed flavours

- Also include fence constructs that model the fences in underlying architectures
  - ISAs need to be able to support these fences
  - Generally speaking, language supports least common denominator of existing ISAs

- Require the program to be data-race-free
  - If not, behaviour is undefined!
Data Races are Evil

- DRF0 is foundation of C++ and Java memory models
  - Compiler may freely re-order between syncs
  - But, it may not introduce any new data races
    - Can’t do speculative loads (can disallow optimizations)

- Why are races so evil?
  - They break reasonable optimization [Boehm and Adve PLDI 2008]:
    ```
    unsigned int i = x;  // x is shared variable
    if (i < 2) {
        // opt: don’t copy x, use by ref
        foo: ...
        // suppose x changes here...
        switch (i) {
        // opt: implement as jump table
            case 0: ... break;
            case 1: ... break;
            default: ... break;  // opt: range inference tells
                // compiler this case is
            }  // opt: range inference tells
        }  // opt: range inference tells
    }  // compiler this case is
    // impossible, drop the check
    ```
C++ Atomics

- Sequentially consistent (SC) atomics have a total order in the program that is respected by all cores
  - e.g., IRIW with all SC atomics is forbidden by C++
- Release and acquire are akin to RC’s release and acquire, but...
  - Releases only synchronize if read by an acquire (corner cases exist with release sequences)
  - Release-acquire synchronization is cumulative (will cover cumulativity next week)
  - Release-acquire accesses do not enforce a total order across all cores
  - IRIW with only release-acquire accesses is allowed by C++
- Relaxed atomics
  - For the case where the programmer just wants a regular load or store without additional added synchronization
  - Conflicting relaxed atomics are NOT data races!
Mapping C++ Atomics to Assembly

- Mapping: set of instruction sequences of a given ISA that are used to implement high-level language synchronization constructs

- ISAs with strong MCMs are easier to map to
  - E.g., In x86, releases and acquires (and relaxed) atomics map to regular loads and stores
  - SC atomics can be implemented by putting an MFENCE after each SC store or a fence before each SC load
    - Why?
Mapping C++ Atomics to Assembly

- Meanwhile, for e.g., ARMv8, more fences are required (or ARM’s acquire-release instructions)
  - Release/acquire atomics can be implemented by ARM’s acquire-release instructions (which work virtually exactly like SC atomics) or by putting fences before releases and after acquires
  - SC atomics can be implemented the same way, though an additional fence in between SC stores and SC loads on the same core is also required
  - Relaxed atomics can map to regular loads and stores
Compiler and Hardware BOTH Play a Role

• If either the compiler or the hardware do not uphold their portion of the ordering requirements, incorrect behaviour can result

• e.g., if an x86 processor doesn’t correctly implement TSO, then C++ programs won’t run correctly, even if the compiler uses a mapping intended for TSO!

• Likewise, if the compiler uses an incorrect mapping, C++ programs also won’t run correctly!

• Need clear and unambiguous specifications of who enforces which orderings

  ☐ If not, can lead to a scenario where heavy-handed fixes are required
The ARM Load-Load Hazard

• ARM ISA spec was ambiguous about whether same-address load-load ordering was maintained
  ○ Natural language again!

• Certain processors assumed it wasn’t required, and certain compilers assumed that hardware would enforce the ordering

• Result: no one maintained the ordering!

• Very hard to fix processors out in the wild

• ARM’s fix: make the compiler insert a fence after every atomic load [ARM 2011]
  ○ Up to a 10% performance degradation [Trippel et al. ASPLOS 2017]
Speculation
The store problem in MP

Uniprocessors: Never wait to write memory

Multiprocessors: Must wait - order matters!

How to know which stores must wait?
The Consistency Debate

Programmer must choose:

**Strict memory ordering**
- e.g., Sequential Consistency (SC)
  - Intuitive – like multitasked uni.
  - Slow – wait for all stores

**Relaxed memory ordering**
- e.g., Sun RMO
  - Complex – SW enforces order
  - Fast – parallel / OoO accesses

Can we get intuitive and fast?
The Big Misconception: Inherent Large Performance Gap

• But, strong ordering thought to hurt performance – One reason for a variety of memory models and flavors

• Not true!

• Memory only has to appear to be ordered
  - Hardware can relax order speculatively
  - Save state while speculating
  - Roll back if relaxed order observed by others
  - Similar to transactional memory, but key difference:
    - Hardware decides the size of “transactions”, not the programmer!
    - E.g. result, SC + Speculation ≥ RC!

• This is the Bart Simpson’s approach to relaxing order: “I didn’t do it. No one saw me doing it!”
Evolution of SC Systems

• Naïve SC – every access is ordered

• Optimized SC
  □ Three simple optimizations
  □ Existing pipeline HW
  □ E.g., MIPS R10K

• No-wait SC
Enhancing SC’s Performance

1. Store buffering
   - Store misses (for either data or write permission) can be removed from pipeline in program order
   - Place them in store buffer

2. All miss “fetches” can be overlapped
   - L1 cache is point of serialization/visibility
   - Requests for cache blocks can be sent out in parallel
   - All accesses to L1 must be atomic and in order
     - Order in which blocks are fetched/filled has no bearing on actual observed program order, until access is performed

3. Load hits can “speculate” past store misses
Execution in strict SC

- Miss on Wr A stalls all unrelated accesses

Memory accesses issue one-at-a-time
SC + Store Buffer

- Removes pending stores from ROB...
- ...but still no memory parallelism
SC + Store Buffer + Prefetching

[Gharachorloo 91]

- Key Idea: Separate fetching write permission from writing to the cache
  - Prefetch performs coherence ops in advance
  - Commit value to cache when consistency requirements fulfilled

- May need to re-request coh. permission upon commit
MIPS R10K:
SC + SB + Prefetch + In-window Load Speculation

[Gharachorloo 91]

- Key Idea: Perform load speculatively, use branch rewind to roll back if the value of the load changes
  - Invalidation messages “snoop” load-store queue
    - If invalidation “hits” a speculative load, rewind & re-execute
    - Alternative implementation – redo all loads in program order at retirement (“Value”-based ordering) [Cain & Lipasti 04]
Memory Ordering Still Causes Stalls

- ... Even with relaxed memory models
  - Memory fences & atomic RMW’s (synchronization)
- ... Even with aggressive in-window speculation
  - Can’t tolerate long miss latencies
How to eliminate waits for stores?

Speculate no one will notice OoO accesses
- Keep going past store misses & fences
- Detect races via coherence protocol

Can outperform RMO!

Requirements
- Scalable rollback support
- Scalable & fast race detection
- Infrequent races

[Gniady 99]
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 1st 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 [Ceze’07])
Roadmap

• InvisiFence for weak ordering
• Generalizing InvisiFence to stronger models
• Subsuming in-window speculation
• Conclusions
Background: Relaxed Consistency

• Relaxes ordering except at programmer-inserted synch.
  • Allows unordered store buffer to hide store misses
• Unordered, coalescing store buffers → 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 Relaxed Consistency

- Add deep speculation to eliminate stalling on fences
  - Post-retirement

- 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: ROB, Head, Tail
- P1: Ckpt
- L1, L2: r/w?
- SB: r/w?
- Dirty

Legend:
- L2
- L1
- ROB
- Head
- Tail
- Ckpt
- Dirty
- r/w?
- S, L, F
- OtherInsn
InvisiFence: Violation Detection

At store retirement:
Set write bit
InvisiFence: Violation Detection

At load retirement:
Set read bit

Key:
- S: Store
- F: Fence
- L: Load
- Other Insn
InvisiFence: Violation Detection

To detect violations: snoop bits
Dirty Head

InvisiFence: Version Management

P0

ROB

Tail

Head

Clean to L2 before 1st speculative write

P1

Key

S Store

F Fence

L Load

Other Insn

Dirty

Dirty

r/w?

r/w?

r

w

r/w?
InvisiFence: Version Management

P0

Tail

ROB

Head

Ckpt

L1

r/w?

SB

r/w?

Dirty

w

Dirty

w

Dirty

w

Dirty

w

Key

S Store

F Fence

L Load

Other Insn
InvisiFence: Version Management

P0

Tail | ROB | Head
--- | --- | ---

P1

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

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

Diagram:
- L1: r/w?
- SB: r/w?
- L2: Dirty
- Ckpt

Legend:
- w: Write
- r: Read
InvisiFence: Rollback

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

```plaintext
P0
Tail | ROB | Head
---|-----|-----

L1
- r/w?
- Dirty: w, r
- Dirty: w

SB
- r/w?
- Ckpt

P1

L2
- Dirty

P1 to P0 Transition:
- Marking the Ckpt
```
InvisiFence: Rollback

- Flash-inval spec. dirty blocks
- Flash-clear bits
- Flash-inval spec. dirty blocks

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

Begin re-execution

Rollback: Fast & simple

Key

S Store
F Fence
L Load
Other Insn
InvisiFence: When to Commit?

Back to speculation: Store returns
InvisiFence: When to Commit?

Move store & r/w bit from SB to L1
InvisiFence: When to Commit?

No outstanding stores: Legal to commit
InvisiFence: Commit

P0

Tail | ROB | Head
--- | --- | ---

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

L2

Dirty

Dirty

Dirty

Dirty

Ckpt

Discard checkpoint

Flash-clear bits

Key

S Store
F Fence
L Load
Other Insn
InvisiFence: Commit

Commit: Fast & simple
InvisiFence Performance

SimFlex simulation of 16-node directory-based SPARC MP
SPARC’s RMO
InvisiFence Performance

13% max speedup; 6% avg

InvisiFence eliminates fence stalls without violations
But what about models requiring stronger ordering?
Generalizing InvisiFence for Strong Ordering

• Strong models impose additional ordering constraints
  • e.g., 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
Violations are negligible (3% slowdown from IF-RMO)

How does this compare to prior work?
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
- Bounded duration to prevent deadlock!
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 relaxed consistency

• 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