EECS 570
Lecture 15
Programming Language
MCMs and Spec. MCM Impl. (contd.)

Winter 2022
Prof. Yatin Manerkar

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

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]:
    ```c
    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
        } //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 – very few orderings (e.g. per-loc SC)
  - 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?
Is there an Inherent Large Performance Gap?

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

• Not necessarily 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!

• Drawback: More complicated implementations

• 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 coher. permission from performing the operation
  - Prefetch does coherence ops in advance
  - Read/commit value to cache when consistency requirements fulfilled

- May need to re-request coher. 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]

<table>
<thead>
<tr>
<th>Store buffer</th>
<th>CPU pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wr A</td>
<td>Rd B</td>
</tr>
<tr>
<td></td>
<td>Wr C</td>
</tr>
<tr>
<td></td>
<td>Rd D</td>
</tr>
<tr>
<td></td>
<td>Rd E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rd E</th>
<th>Not fetched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd D</td>
<td>Miss</td>
</tr>
<tr>
<td>Wr C</td>
<td>Prefetch</td>
</tr>
<tr>
<td>Rd B</td>
<td>Complete</td>
</tr>
<tr>
<td>Wr A</td>
<td>Miss</td>
</tr>
</tbody>
</table>
Memory Ordering Still Causes Stalls

- ... Even with relaxed memory models
  - Memory fences & atomic RMWs (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 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 [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**: Other instructions
InvisiFence: Violation Detection

At store retirement:
Set write bit

Key
- S: Store
- F: Fence
- L: Load
- OtherInsn

Diagram shows a system with ROB, Head, Tail, and a checkpoint (Ckpt). The diagram includes L1, L2, and an r/w? indication for each level.
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: Other Insn
InvisiFence: Version Management

Clean to L2 before 1st speculative write

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

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

Legend:
- Dirty
- r/w?
InvisiFence: Version Management

Key:
- S: Store
- F: Fence
- L: Load
- OtherInsn: Other Instructions
InvisiFence: Version Management

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

Key:
- S: Store
- F: Fence
- L: Load
- OtherInsn

Diagram:
- L1: r/w?
- SB: r/w?
- L2: Dirty
- P0: Tail ROB Head
- P1: Ckpt
- Dirty w
- Dirty r
- Dirty w
- Dirty w
- Dirty w
InvisiFence: Rollback

Restoring checkpoint

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

Key:
- S: Store
- F: Fence
- L: Load
- OtherInsn
InvisiFence: Rollback

Rollback: Fast & simple

Begin re-execution

Key

- S: Store
- F: Fence
- L: Load
- OtherInsn
InvisiFence: When to Commit?

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

P0

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

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

Dirty  | w | Dirty  | r | Dirty  | w

Move store & r/w bit from SB to L1

L2  | Dirty

Key

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

No outstanding stores: Legal to commit
InvisiFence: Commit

P0
Tail | ROB | Head
L1  | r/w?| SB  | r/w?

L2
Dirty

P1

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