EECS 570
Lecture 16
Relaxed Consistency

Winter 2018

Prof. Satish Narayanasamy

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

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
  - E.g. result, SC + Speculation \( \geq \) 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

• Wait-free 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

<table>
<thead>
<tr>
<th></th>
<th>Wr A</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd B</td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>Wr C</td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>Rd D</td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td>Rd E</td>
<td></td>
<td>Not fetched</td>
</tr>
</tbody>
</table>
SC + Store Buffer + Store Prefetching

[Gharachorloo 91]

- Key Idea: Separate fetching write permission from writing to the cache
  - “Store prefetch” performs coherence ops in advance
  - Commit value to cache when write leaves ROB
- May need to re-request store 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 complete 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
  - Frequent memory fences & atomic RMW’s (synchronization)
- ... Even with aggressive in-window speculation
  - Can’t tolerate long miss latencies
How to be store-wait free?

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]
Early HW solutions
[Ranganathan 97] [Gniady 99]

- Log all instructions
  - Large storage requirement
- Read old value before store
  - Extra L1 traffic
- Assoc. search on external req.
  - Limited capacity

Early solutions require impractical mechanisms
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: Weak Ordering

• Relaxes ordering except at programmer-inserted fences
  - 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 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
InvisiFence: Violation Detection

At store retirement:
Set write bit

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

L1: r/w?
SB: r/w?
L2: 

P0
Tail ROB Head

P1

Dirty
Dirty
Dirty

L1 S LS

Ckpt

r/w?
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

L1 r/w? SB r/w?
Dirty
Dirty

L2

Key
S Store
F Fence
L Load
Other Insn
**InvisiFence: Version Management**

**P0**

- **Tail**
- **ROB**
- **Head**
- **Ckpt**

**L1**
- r/w?

**SB**
- r/w?

- **Dirty**
- **r**
- **w**

**L2**
- **Dirty**

**P1**

Clean to L2 before 1\textsuperscript{st} speculative write

**Key**

- **S** Store
- **F** Fence
- **L** Load
- **Other Insn**
InvisiFence: Version Management

**P0**

- Tail
- ROB
- Head
- Ckpt

**P1**

- L1
- SB

**Key**
- Store
- Load
- Fence
- Other Insn
InvisiFence: Version Management

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

P0

Tail ROB Head

L1 r/w? SB r/w?

Dirty w
Dirty w
Dirty w
Dirty w

L2

Dirty

P1

Key

Store
Load
Fence
Other Insn
InvisiFence: Rollback

L0
Tail  ROB  Head
Ckpt
P1

L1  r/w?
Dirty  w
Dirty  r
Dirty  w

SB  r/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

Begin re-execution

Rollback: Fast & simple

Key

- Store
- Load
- Fence
- Other Insn
InvisiFence: When to Commit?

Back to speculation: Store returns

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

P0

Tail
ROB
Head

Ckpt

L1 r/w? SB r/w?

Dirty w
Dirty
Dirty r
Dirty w

L2

Dirty

Move store & r/w bit from SB to L1

Key

Store
Load
Fence
Other Insn
InvisiFence: When to Commit?

L0

Tail    ROB    Head

Ckpt

L1  r/w?

Dirty  w

Dirty  b

r

r

Dirty  w

L2

Dirty

No outstanding stores:
Legal to commit

Key:

Store

Load

Fence

Other Insn
InvisiFence: Commit

P0

Tail ROB Head

P1

Discard checkpoint

Flash-clear bits

L1 r/w?

SB r/w?

L2

Dirty w

Dirty

Dirty r

Dirty w

Key

Store Load Other Insn

Fence
InvisiFence: Commit

Commit: Fast & simple
InvisiFence Performance

SimFlex simulation of 16-node directory-based SPARC MP
SPARC’s RMO (similar to Alpha, ARM, PowerPC)
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
  - 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 vs Violation, SB drain, SB full, Other for benchmarks]

- Violation
- SB drain
- SB full
- Other

Benchmarks: apache, zeus, oracle oltp-db2, dss-db2, barnes, ocean
Continuous Speculation Performance

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
Review: Problems with SC

• Difficult to implement efficiently in hardware
  ❑ Straight-forward implementations:
    ☐ No concurrency among memory access
    ☐ Strict ordering of memory accesses at each node
    ☐ Essentially precludes out-of-order CPUs

• Unnecessarily restrictive
  ❑ Most parallel programs won’t notice out-of-order accesses

• Conflicts with latency hiding techniques
Execution in strict SC

- Miss on Wr A stalls all unrelated accesses

Memory accesses issue one-at-a-time
Relaxing Write-to-Read Order

- Motivation: Post-retirement store buffers
  - Allow reads to bypass incomplete writes
    - Reads search store buffer for matching values
    - Hides all latency of store misses in uniprocessors
  - Writes are still ordered w.r.t. other writes
  - Reads are still ordered w.r.t. other reads
Dekker's Algorithm w/ Store Buffer

P1
A = 1;
if (B != 0) goto retry;
/* enter critical section*/

P2
B = 1;
if (A != 0) goto retry;
/* enter critical section*/
Processor Consistency (PC)

• Formal requirements [Goodman 1989]:
  ❑ Before LOAD is performed w.r.t. any other processor, all prior LOADs must be performed
  ❑ Before STORE is performed w.r.t. any other processor, all prior mem ops must be performed

• Does not require store atomicity
• This is basically what x86 and VAX do
• Also roughly what IBM’s System-370 did (1960’s)
  ❑ Oddly, on 370, loads must stall if they hit in the SB
Sun’s “Total Store Order” (TSO)

• Formal requirements [v8 architecture manual]:
  ▶ **Order** - There exists a partial memory order (<M) and it is *total* for all operations with store semantics
  ▶ **Atomicity** - Atomic read-modify-write-writes do not allow a store between the read and write parts
  ▶ **Termination** - All stores are performed in finite time
  ▶ **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  ▶ **LoadOP** – Loads are blocking
  ▶ **StoreStore** – Stores are ordered

• This is the same as PC, except that it requires store atomicity
PC/TSO: Programmer’s Perspective

- Can occasionally lead to astonishing behavior changes
  - E.g., Dekker’s algorithm doesn’t work
  - ISAs provide an STBAR (store barrier) to manually force order
    - Semantics – store buffer must be empty before memory operations after STBAR may be executed
  - Can also enforce order by replacing stores with RMWs

- But, the key case, where sync is done with locks, simply works
  - Lock acquires are RMW operations \(\Rightarrow\)
    - they force order for preceding/succeeding loads/stores
      - Load semantics of RMW imply load-load orderings
      - Ditto for store semantics
  - Lock release is a store operation \(\Rightarrow\)
    - it must be performed after critical section is done
PC/TSO: Compiler’s Perspective

• Compiler may now hoist loads across stores
  ❑ Still can’t reorder stores or move loads across loads
  ❑ Not clear how helpful this is in practice...
  ❑ ...Recent results from Prof. Satish’s group:
    ❙ 5-30% perf. gap vs. compiler that preserves SC [PLDI’11]

• No new crazy memory barriers to emit
  ❑ Library/OS writers need to use a little bit of caution;
    use RMWs instead of loads/stores for synchronization
  ❑ TSO-unsafe code is rare enough that it can be the
    programmer’s problem
• No need to invoke “undefined behavior” to avoid onerous
  implementation requirements
PC/TSO: HW Perspective

• Allows a **FIFO-ordered, non-coalescing store buffer**
  ❑ Typically maintains stores at word-granularity
  ❑ Loads search buffer for matching store(s)
    ○ Some ISAs must deal with merging partial load matches
  ❑ Coalescing only allowed among adjacent stores to same block
  ❑ Must force buffer to drain on RMW and STBAR
  ❑ Often, this is implemented in same HW structure as (speculative) store queue

• Can hide store latency!
  ❑ But, store buffer may need to be quite big
    ○ Stores that will be cache hits remain buffered behind misses
  ❑ Associative search limits scalability
    ○ E.g., certainly no more than 64 entries
Execution in PC / TSO

- Stores misses do not stall retirement
  - St A, St C, Ld D misses overlapped
  - Ld B retired without waiting for St A to fill
  - St C consumes space in store buffer even though it will hit

PC/TSO hides store miss latency
Relaxing Write-to-Write Order

- Motivation: Coalescing store buffers & early drain

- Allows writes to coalesce in SB & drain early

- Diagram showing coalescing and early drain in the CPU pipeline.
Sun’s “Partial Store Order” (PSO)

• Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is total for all operations with store semantics
  - **Atomicity** - Atomic read-modify-write-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered only if they are separated by a *membar* (memory barrier) instruction
  - **StoreStoreEq** – Stores to the same address are ordered
PSO: Compiler/HW Perspective

• Allows an unordered, coalescing post-retirement store buffer
  ▶ Can now use a cache-block-grain set-associative structure
  ▶ Store misses leave store buffer upon cache fill
  ▶ Loads search buffer for matching store(s)
  ▶ Must force buffer to drain on STBAR

• Much more efficient store buffer

• But, still doesn’t allow out-of-order loads
  ▶ No OoO execution (without speculation)
  ▶ Compiler’s hands are still tied
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 \[\downarrow\downarrow\downarrow\downarrow\]\ 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
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 \textit{minimum} barriers is harder
  - E.g., lock and unlock (from v9 ref. manual; w/o branch delays)

\begin{verbatim}
LockWithLDSTUB(lock)
    retry:    ldstub [lock],%10
              tst %10
              be out
    loop:     ldub [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]
\end{verbatim}
• 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

```
Read / Write
  ...
  Read/Write

Synch

Read / Write
  ...
  Read/Write

Synch

Read / Write
  ...
  Read/Write

Synch

Read / Write
  ...
  Read/Write
```
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 of Release Consistency (RC)](image)

Weak Consistency (WCsc)

Release Consistency (RCpc)

\[ u \text{ cannot perform with respect to any other processor until } u \text{ is performed} \]
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
WO/RC: Compiler’s Perspective

- 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:
    ```
    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
    ```