Announcements

- No Lecture Monday – Jury Duty!
- Project Milestone 2
  - Due 3/22
  - Meetings 3/23 & 3/24
  - Prepare a brief slide deck in lieu of a written report
  - Submit via Canvas
- Programming Assignment 2
  - Due 3/27

Readings

For today:
- Gharachorloo et al - Two Techniques to Enhance the performance of Memory Consistency Models - ICPP 1991

For Wednesday 3/22:
- B. Choi et al, DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism, PACT 2011

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

```
  1. ac 1
  2. lo 1
  3. lo 2
  4. rs 1
  5. rs 2

Release Consistency (RC)
```

```
  1. ac
  2. lo
  3. lo
  4. rs
  5. rs

Weak Consistency (WC)
```

- v cannot perform with respect to any other processor until u 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) {
        foo: ...            // suppose x changes here...
        switch (i) {
          case 0: ... break;
          case 1: ... break;
          default: ... break;  // compiler this case is impossible, drop the check
        }
      }
      ```

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?

Evolution of SC Systems

- Naive SC – every access is ordered

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

- Wait-free SC

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 ≥ RC!

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

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 + Store Prefetching

- 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:

- 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

**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

---

**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 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

At load retirement:
Set read bit

To detect violations:
snoop bits
Clean to L2 before 1st speculative write

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

- **Restore checkpoint**: P0 restores the checkpoint and begins re-execution.
- **Begin re-execution**: P0 starts from the checkpoint.
- **Rollback: Fast & simple**: The rollback process is quick and straightforward.

InvisiFence: When to Commit?

- **Back to speculation**: P1 returns to speculation after the commit.
- **Store returns**: Store returns from the committed state.
- **Move store & r/w bit from SB to L1**: The store and r/w bit are moved from the Store Buffer (SB) to the L1 cache.
InvisiFence: When to Commit?

- **P0**: Tail, ROB, Head
  - L1: r/w?
  - SB: r/w?
  - No outstanding stores: Legal to commit

- **P1**: Ckpt

InvisiFence: Commit

- **P0**: Tail, ROB, Head
  - L1: r/w?
  - SB: r/w?
  - Commit: Fast & simple

- **P1**: Ckpt

InvisiFence Performance

- Normalized Runtime
- Violation
- SB drain
- SB full
- SB full
- Other

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

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
Continuous Speculation Performance

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

IF-cont+ (with commit on violation) achieves IF-sc performance without LSQ snooping