EECS 570
Lecture 16
Speculation & Consistency

Winter 2019
Prof. Thomas Wenisch

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

Announcements

- Last call for exam regrade requests!
- No lecture next Monday - ISPASS
- Project Milestone 2
  - Slides due 3/27
  - Meetings 3/29
  - Prepare a brief slide deck in lieu of a written report
  - Submit via Canvas

- Programming Assignment 2
  - Due 4/1
Readings

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

For Wednesday 3/27:
RCU in a nutshell

- Think about data structures that are mostly read, occasionally written
  - Like the Linux dcache
- RW locks allow concurrent reads
  - Still require an atomic decrement of a lock counter
  - Atomic ops are expensive
- Idea: Only require locks for writers; carefully update data structure so readers see consistent views of data
Motivation
(from Paul McKenney’s Thesis)

Performance of RW lock only marginally better than mutex lock
Principle (1/2)

- Locks have an acquire and release cost
  - Substantial, since atomic ops are expensive
- For short critical regions, this cost dominates performance
Principle (2/2)

- Reader/writer locks may allow critical regions to execute in parallel
- But they still serialize the increment and decrement of the read count with atomic instructions
  - Atomic instructions performance decreases as more CPUs try to do them at the same time
- The read lock itself becomes a scalability bottleneck, even if the data it protects is read 99% of the time
RCU: Split the difference

特派员 One of the hardest parts of lock-free algorithms is concurrent changes to pointers

特派员 So just use locks and make writers go one-at-a-time

特派员 But, make writers be a bit careful so readers see a consistent view of the data structures

特派员 If 99% of accesses are readers, avoid performance-killing read lock in the common case
Example: Linked lists

This implementation needs a lock

Reader goes to B

B's next pointer is uninitialized; Reader gets a page fault
Example: Linked lists

Reader goes to C or B---either is ok

Garbage collect C after all readers finished
Part of what makes this safe is that we don’t immediately free node C

- A reader could be looking at this node
- If we free/overwrite the node, the reader tries to follow the ‘next’ pointer
- Uh-oh

How do we know when all readers are finished using it?

- Hint: No new readers can access this node: it is now unreachable
Quiescence

- Trick: Linux doesn’t allow a process to sleep while traversing an RCU-protected data structure
  - Includes kernel preemption, I/O waiting, etc.
- Idea: If every CPU has called schedule() (quiesced), then it is safe to free the node
  - Each CPU counts the number of times it has called schedule()
  - Put a to-be-freed item on a list of pending frees
  - Record timestamp on each CPU
  - Once each CPU has called schedule, do the free
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 ≥ 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
**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
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

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

At store retirement:
Set write bit
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
InvisiFence: Version Management

Clean to L2 before 1st speculative write
InvisiFence: Version Management
InvisiFence: Version Management

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

P0

Tail
ROB
Head

Ckpt

L1
r/w?

SB
r/w?

Dirty

Dirty

Dirty

w
r
w

L2

Dirty

Key

S Store
F Fence
L Load
Other Insn
InvisiFence: Rollback

Flash-inval spec. dirty blocks

Head
ROB
Tail

P0

Restore checkpoint

L1 r/w?

SB r/w?

L2

Flash-clear bits
Flash-inval spec. dirty blocks

r
w
w
r
w

Dirty
Dirty
Dirty

Key

S Store
F Fence
L Load
Other Insn
InvisiFence: Rollback

Begin re-execution

Rollback: Fast & simple

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

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

**P0**

- Tail
- ROB
- Head

**P1**

- Ckpt

Discard checkpoint

**L1**

- r/w?
- Dirty
- Dirty
- Dirty
- Dirty

Flash-clear bits

**L2**

- Dirty

**Key**

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

Dirty
InvisiFence: Commit

Commit: Fast & simple
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
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
To reduce rollbacks: “Commit on Violation”

- Temporarily defer conflicting requests
Continuous Speculation Performance

Normalized Runtime vs Violation, SB drain, SB full, Other for various benchmarks:
- apache
- zeus
- oracle oltp-db2
- dss-db2
- barnes
- ocean

Legend:
- Violation
- SB drain
- SB full
- Other
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