EECS 570
Lecture 15
SC-Preserving Hardware
Winter 2020
Prof. Satish Narayanasamy

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

“I didn’t do it. Nobody saw me do it. You can’t prove anything”
SC Preserving Hardware
Review: Coherence

A Memory System is Coherent if

- can serialize all operations to that location such that,
- operations performed by any processor appear in program order (<p)
- value returned by a read is value written by last store to that location

There is broad consensus that coherence is a good idea.

But, that is not enough for consistency...
Coherence vs. Consistency

A=0  flag=0
Processor 0
A=1;
flag=1;

Processor 1
while (!flag); // spin
print A;

• Intuition says: P1 prints A=1

• Coherence says: absolutely nothing
  □ P1 can see P0’s write of flag before write of A!!! How?
    ● P0 has a coalescing store buffer that reorders writes
    ● Or out-of-order execution
    ● Or compiler re-orders instructions

• Imagine trying to figure out why this code sometimes “works” and sometimes doesn’t

• Real systems act in this strange manner
  □ What is allowed is defined as part of the ISA of the processor
Coherence vs. Consistency

Coherence is not sufficient to guarantee a memory model

Coherence concerns only one memory location
Consistency concerns apparent ordering for all locations

Coherency = SC for accesses to one location
- Guarantees a total order for all accesses to a location that is consistent with the program order
  - value returned by a read is value written by last store to that location
Tools to reason about memory models

• Time? Generally impractical, but may be useful for some systems and use cases (e.g., Lamport clocks)

• (Partially) ordered sets
  - A → B ∧ B → C ⇒ A → C (transitive)
  - A → A (reflexive)
  - A → B ∧ B → A ⇒ A = B (antisymmetric)

• Some important (partial) orders
  - Program order (<p) – per-thread order in inst. sequence
  - Memory order (<M) – order memory ops are performed
When is a mem. op. “performed”?

- Nuanced definitions due to [Scheurich, Dubois 1987]
  - A Load by $P_i$ is performed with respect to $P_k$ when new stores to the same address by $P_k$ can not affect the value returned by the load.
  - A Store by $P_i$ is performed with respect to $P_k$ when a load issued by $P_k$ to the same address returns the value defined by this (or a subsequent) store.
  - An access is performed when it is performed with respect to all processors.
  - A Load by $P_i$ is globally performed if it is performed and if the store that is the source of the new value has been performed.
SC: Hardware

• Formal Requirements:
  - Before LOAD is performed w.r.t. any other processor, all prior LOADs must be globally performed and all prior STOREs must be performed.
  - Before STORE is performed w.r.t. any other processor, all prior LOADs globally performed and all previous STORE be performed.
  - Every CPU issues memory ops in program order.

• In simple words:
  SC: Perform memory operations in program order.
Sequential Consistency (SC)

- Processors appear to perform memory ops in program order.
- After each memory op, the switch is randomly set.
- Provides total order among all operations.
Sufficient Conditions for SC

“A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”

-Lamport, 1979

Every proc. “performs” memory ops in program order

One implementation:

Memory ops happen (start and end) atomically

- Each processor core waits for a memory access to complete before issuing next memory op

Easily implemented with a shared bus
Dekker’s Algorithm

- Mutually exclusive access to a critical region
  - Works as advertised under sequential consistency

```c
/* initial A = B = 0 */

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

P2
B=1;
if (A != 0) goto retry;
/* enter critical section*/
```
Problems with SC Memory Model

• 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
E.g., Add a Store Buffer

- Allow reads to bypass incomplete writes
  - Reads search store buffer for matching values
  - Hides all latency of store misses in uniprocessors, but...
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*/
Naïve SC Processor Design

Requirement: Perform memory operations in program order

Assume

- coherence
- store atomicity
- + memory ordering restrictions

Memory ordering restrictions

- Processor core waits for store to complete, before issuing next memory op
- Processor core waits for load to complete, before issuing next op
Store Atomicity

- **Store atomicity** – property of a memory model stating the existence of a total order of all state-changing memory ops.
  
  - What does this mean?
    - All nodes will agree on the order that writes happen

<table>
<thead>
<tr>
<th>A=0</th>
<th>B=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1 A=1;</td>
<td>P2 B = 1;</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Under store-atomicity, what results are (im-)possible?
Implementing Store Atomicity

- On a bus...
  - Trivial (mostly); store is globally performed when it reaches the bus

- With invalidation-based directory coherence...
  - Writer cannot reveal new value till all invalidations are ack’d

- With update-based coherence...
  - Hard to achieve... updates must be ordered across all nodes

- With multiprocessors & shared caches
  - Cores that share a cache must not see one another’s writes! (ugly!)
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 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

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

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>Idle</td>
<td></td>
</tr>
<tr>
<td>Wr C</td>
<td>Idle</td>
<td></td>
</tr>
<tr>
<td>Rd D</td>
<td>Idle</td>
<td></td>
</tr>
<tr>
<td>Rd E</td>
<td>Not fetched</td>
<td></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
SC Hardware Overhead

- Memory operations in pipeline cannot be executed out-of-order
- Speculative execution of loads in execution window [Gharachorloo et al., 1991]
- Loads must wait for store buffer drain
- Stores must retire in-order
- Speculative load commit [Ranganathan et al., 1997]
- Speculative store commit [Gniady et al., 1999]
- Require complex checkpoint and recovery support beyond execution window!!
Opportunity

• *Safe* and *Unsafe* accesses
  – Private or read-only shared accesses are *safe*

• No need to enforce memory model constraints for *safe* accesses  
  [Shasha & Snir, 1988; Adve, 1993]

• Large fraction of memory accesses are safe  
  [Hardvellas et al., 2009; Cuesta et al., 2011]
Conventional SC

- Loads must wait for store buffer drain
- Stores must retire in-order

Access Type Aware SC

- Unsafe loads must wait for unsafe store buffer drain
- Unsafe stores must retire in-order

ROB

- Ld Z

Store buffer

- St X, St Y

Unsafe store buffer

- Ld Z
- y

Safe Store buffer

- St Y
- St X

→ St Y
→ St X
→ Ld Z
Two Store Buffers: Correctness Challenge

- **Uniform-Type assumption**
  - All accesses to a memory location are of the same type

- **Store-to-load forwarding**
  - Safe loads look-up only safe store buffer and *vice-versa*

- **Store-to-store program order**
  - Stores to a location will be committed into same store buffer

- **Challenge:**
  - Safe/unsafe classifier may transiently violate Uniform-Type assumption
  - Ensure correct store-to-load and store-to-store semantics
Identifying Safe Accesses

• Static classification scheme

• Dynamic classification scheme

• Hybrid = Static + Dynamic
Static Classifier

• Conservative analysis to identify *safe* accesses
  – Non-escaping function locals, temporaries, and literals

• ISA is extended to indicate type

• Safe bit is set at decode
Two Store Buffers: Correctness Challenge for Static Classifier

- **Uniform-type assumption**
  - All accesses to a memory location are of the same type (safe or unsafe)

- **Violation #1**: Mixed Access
  - A memory instruction that can access variables of different types

- **Violation #2**: Stack location reuse across different function calls
Dynamic Classifier

- Dynamically classify pages as safe or unsafe
  - Extend page protection to thread level
  - Extend TLB to track page type

[Dynamic Classifier Figure]

Safe bit set during address translation

[Dunlap et al. VEE’08]
Dynamic Classifier: Ensuring Correctness

• **Problem:** A page’s type can change from safe to unsafe
  – Access-type assumption may be violated
  – SC may be violated

• **Solution:**
  – Drain store buffers of processors that last accessed the page
Hybrid Classification

• Access is safe if
  statically safe OR dynamically safe

• Reason: Both static and dynamic schemes are conservative
SC-Hardware Overhead

Average overhead for SC: 10.2% ➔ 7.5% ➔ 5.3% ➔ 4.5%
Average performance cost of end-to-end SC is 7.5% w.r.t stock compiler on TSO.