Hardware Memory Models: x86-TSO

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu
Agenda

• So far
  — hardware organization
    – multithreading
    – cores
    – caches
    – multisocket systems
  — overview of memory models

• Today: hardware implementation of TSO memory model
Requirement for Multithreaded HW

Reliable, high-performance parallel code

• Operating system kernel

• Libraries
  — language runtime systems
  — synchronization primitives
  — concurrent data structures

• Compilers for concurrent languages
Challenges - I

• Multiprocessors typically do not provide sequentially consistent memory
  — why? optimizations for performance!
    – e.g., store buffers to hide write latency, speculative execution

• Multithreaded codes observe relaxed memory model
  — threads have only loosely consistent views of shared memory
    – e.g., visible evidence of store buffering on x86

• Understanding relaxed model is necessary for writing correct parallel software

Figure credit: [1]

Figure credit: [2]
Store Buffers and Store Forwarding

- Problem: without store forwarding a memory location may appear to have multiple values

Figure 5: Caches With Store Buffers

Figure 6: Caches With Store Forwarding

Figure credit: [2]
Challenges - II

- Different processor families use different relaxed models
- Commodity vendors often specify memory models in ambiguous, informal prose
  - Poor medium for loose specifications
    - Inevitably ambiguous
    - Sometimes wrong
    - No examples
  - Cause for confusion: spin lock optimization? (LKM 1999)
- Major and subtle differences between processor families
  - What non-SC behaviors they permit
  - Memory barriers and synchronization instructions they provide
Architectural Specifications

- Specify what programmers can rely upon
- Architectural specifications are “loose” to cover past and future implementations

Behaviors permitted by today’s systems

Behaviors permitted by a HW Memory Model
Memory Models for x86 Processors

- Problem: some prior Intel and AMD specifications
  - contain serious ambiguities
  - are arguably too weak for writing programs
  - are simply unsound with respect to actual hardware
  - provide no basis for formally reasoning about programs

- Contribution: new x86-TSO programmer’s model
  - TSO = total store order
  - suffers from none of the aforementioned problems
  - provides intuitive abstract machine, accessible to programmers
  - is mathematically precise: rigorously defined in HOL4
    - memory model + semantics for machine instructions enables formal reasoning about program behavior

- How is this useful?
  - guides intuition of systems programmers developing software for multithreaded systems
Causal Consistency (IWP/AMD3.14/x86-CC)

- **Definition:** memory ordering obeys causality
  - respects transitive visibility

- **Problem:** causal consistency is too weak for programmers
  - admits the following inconsistent view of independent writes

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
<td>MOV EAX ← [x]</td>
<td>MOV ECX ← [y]</td>
</tr>
<tr>
<td>MOV EBX ← [y]</td>
<td>MOV EDX ← [x]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Forbidden Final State:** Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0

**cc:** Allow; **tso:** Forbid

- ordering inconsistencies can arise if store buffers are shared between some but not all threads
- would need to use LOCK instead of MFENCE to recover SC
- appears looser than behavior of implemented processors

Figure credit: [1]
x86 Fences

• Definitions
  — LFENCE: load fence
  — SFENCE: store fence
  — MFENCE: memory fence (strongest x86 memory barrier)

• Operation
  — reads cannot pass LFENCE and MFENCE instructions
  — writes cannot pass SFENCE and MFENCE instructions
Problem 2: unsound with respect to current processors

- Core 2 Duo allows behavior below, although disallowed by x86-CC

---

### Table 1

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 2</td>
</tr>
<tr>
<td>MOV EAX ← [x]</td>
<td>MOV [x] ← 2</td>
</tr>
<tr>
<td>MOV EBX ← [y]</td>
<td></td>
</tr>
</tbody>
</table>

Allowed Final State: Proc 0:EAX = 1 ∧ Proc 0:EBX = 0 ∧ [x] = 1

---

**how?**

- P1 write of [y] = 2 is buffered
- P0 buffers its write of [x] = 1, reads [x] = 1 from its store buffer, and reads [y] = 0 from main memory
- P1 buffers its [x] = 2 write, flushes its buffered [y] = 2, [x] = 2 writes to memory
- P0 flushes its [x] = 1 write to memory.
Causal Consistency

- Processes in a system agree on the relative ordering of operations that are causally related

- Defined by
  - program order
  - writes into order
Causal Consistency

Because there may be multiple writes of a value to a location, there may be more than one writes-into order. A writes-into order \( \Rightarrow \) on \( H \) is any relation with the following properties:

- if \( o_1 \Rightarrow o_2 \), then there are \( x \) and \( v \) such that \( o_1 = w(x)v \) and \( o_2 = r(x)v \);
- for any operation \( o_2 \), there is at most one \( o_1 \) such that \( o_1 \Rightarrow o_2 \);
- if \( o_2 = r(x)v \) for some \( x \) and there is no \( o_1 \) such that \( o_1 \Rightarrow o_2 \), then \( v = \perp \); that is, a read with no write must read the initial value.

A causality order \( \rightsquigarrow \) induced by \( \Rightarrow \) for \( H \) is a partial order that is the transitive closure of the union of the history’s program order and the order \( \Rightarrow \). In other words, \( o_1 \rightsquigarrow o_2 \) if and only if one of the following cases holds:

- \( o_1 \Rightarrow o_2 \) for some \( p_i \) (\( o_1 \) precedes \( o_2 \) in \( L_i \));
- \( o_1 \Leftrightarrow o_2 \) (\( o_2 \) reads the value written by \( o_1 \)); or
- there is some other operation \( o' \) such that \( o_1 \rightsquigarrow o' \rightsquigarrow o_2 \).

(If the relation \( \rightsquigarrow \) is cyclic, then it is not a causality order.) If \( o_1 \) and \( o_2 \) are two operations in \( H \) such that, for causality order \( \rightsquigarrow \), \( o_1 \nrightarrow o_2 \) and \( o_2 \nrightarrow o_1 \), we say that \( o_1 \) and \( o_2 \) are concurrent with respect to \( \rightsquigarrow \).
Causal Consistency (IWP/AMD3.14/x86-CC)

- Problem 2: unsound with respect to current processors
  - Core 2 Duo allows behavior below, although disallowed by x86-CC

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 2</td>
</tr>
<tr>
<td>MOV EAX ← [x]</td>
<td>MOV [x] ← 2</td>
</tr>
<tr>
<td>MOV EBX ← [y]</td>
<td></td>
</tr>
</tbody>
</table>

Allowed Final State: Proc 0:EAX=1 ∧ Proc 0:EBX=0 ∧ [x]=1

- how?
  - P1 write of [y]=2 is buffered
  - P0 buffers its write of [x]=1, reads [x]=1 from its store buffer, and reads [y]=0 from main memory
  - P1 buffers its [x]=2 write, flushes its buffered [y]=2, [x]=2 writes to memory
  - P0 flushes its [x]=1 write to memory.

Figure credit: [1]
x86-TSO Programmer’s Model

- Store buffers are FIFO, a reading thread must read its most recent buffered write, or if none present, value from memory.
- MFENCE flushes a thread’s store buffer.
- LOCK’d instruction (LOCK is a modifier that applies to other instructions)
  - thread must obtain global lock
  - after instruction, thread flushes its store buffer
  - no other thread can read while global lock is held.
- A buffered write can propagate to memory at any time, except when another thread holds the global lock.
Scope of x86-TSO

• Programs using cacheable, write-back memory

• Without

  — exceptions
  — misaligned accesses
  — non-temporal operations (which avoid updating L1 cache)
  — self-modifying code
  — page table changes
Example 8–1. Stores Are not Reordered with Other Stores.

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV EAX ← [y]</td>
</tr>
<tr>
<td>MOV [y] ← 1</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

Forbidden Final State: Proc 1:EAX=1 ∧ Proc 1:EBX=0

This test implies that the writes by Proc 0 are seen in order by Proc 1’s reads, which also execute in order. x86-TSO forbids the final state because Proc 0’s store buffer is FIFO, and Proc 0 communicates with Proc 1 only through shared memory.
**Example 8-2. Stores Are Not Reordered With Older Loads.**

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV EAX ← [x]</td>
<td>MOV EBX ← [y]</td>
</tr>
<tr>
<td>MOV [y] ← 1</td>
<td>MOV [x] ← 1</td>
</tr>
</tbody>
</table>

Forbidden Final State: Proc 0: EAX=1 ∧ Proc 1: EBX=1

x86-TSO forbids the final state because reads are never delayed.
**Example 8–3. Loads May Be Reordered with Older Stores.** This test is just the SB example from Section 1, which x86-TSO permits.

<table>
<thead>
<tr>
<th>SB</th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td></td>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
<tr>
<td>Allowed Final State:</td>
<td>Proc 0:EAX=0 ∧ Proc 1:EBX=0</td>
<td></td>
</tr>
</tbody>
</table>

**Example 8–5. Intra-Processor Forwarding Is Allowed.** This test is similar to Example 8–3.
**Example 8-4. Loads Are not Reordered with Older Stores to the Same Location.**

<table>
<thead>
<tr>
<th>Proc 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
</tr>
<tr>
<td>MOV EAX ← [x]</td>
</tr>
<tr>
<td>Required Final State: Proc 0: EAX = 1</td>
</tr>
</tbody>
</table>

x86-TSO requires the specified result because reads must check the local store buffer.
Example 8–6. Stores Are Transientely Visible.

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV [x]←1</strong></td>
<td><strong>MOV EAX←[x]</strong></td>
<td><strong>MOV EBX←[y]</strong></td>
</tr>
<tr>
<td></td>
<td><strong>MOV [y]←1</strong></td>
<td><strong>MOV ECX←[x]</strong></td>
</tr>
</tbody>
</table>

Forbidden Final State: Proc 1:EAX=1 ∧ Proc 2:EBX=1 ∧ Proc 2:ECX=0

x86-TSO forbids the given final state because otherwise the Proc 2 constraints imply that y was written to shared memory before x. Hence the write to x must be in Proc 0’s store buffer (or the instruction has not executed), when the write to y is initiated. Note that this test contains the only mention of “transitive visibility” in the Intel SDM, leaving its meaning unclear.
Example 8-7. Stores are seen in a consistent order by other processors. This test rules out the IRIW behavior as described in Section 2.2. x86-TSO forbids the given final state because the Proc 2 constraints imply that x was written to shared memory before y whereas the Proc 3 constraints imply that y was written to shared memory before x.

IRIW

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
</table>


Figure credit: [1]
**Example 8–8.** **Locked Instructions Have a Total Order.** This is the same as the IRIW Example 8–7 but with LOCK’d instructions for the writes; x86-TSO forbids the final state for the same reason as above.

**IRIW**

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOV [x] ← 1</strong></td>
<td><strong>MOV [y] ← 1</strong></td>
<td><strong>MOV EAX ← [x]</strong></td>
<td><strong>MOV ECX ← [y]</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>MOV EBX ← [y]</strong></td>
<td><strong>MOV EDX ← [x]</strong></td>
</tr>
</tbody>
</table>


Figure credit: [1]
**Example 8–9. Loads Are not Reordered with Locks.**

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCHG [x] ← EAX</td>
<td>XCHG [y] ← ECX</td>
</tr>
<tr>
<td>MOV EBX ← [y]</td>
<td>MOV EDX ← [x]</td>
</tr>
</tbody>
</table>

Initial state: Proc 0:EAX = 1 \(\land\) Proc 1:ECX = 1 (elsewhere 0)

Forbidden Final State: Proc 0:EBX = 0 \(\land\) Proc 1:EDX = 0

This test indicates that locking both writes in Example 8–3 would forbid the nonsequentially consistent result. x86-TSO forbids the final state because LOCK’d instructions flush the local store buffer. If only one write were LOCK’d (say the write to x), the Example 8–3 final state would be allowed as follows: on Proc 1, buffer the write to y and execute the read x, then on Proc 0 write to x in shared memory then read from y.
Example 8–10. Stores Are not Reordered with Locks. This is implied by Example 8–1, as we treat the memory writes of LOCK’d instructions as stores.

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV [x] ← 1</td>
<td>MOV [y] ← 1</td>
</tr>
<tr>
<td>MFENCE</td>
<td>MFENCE</td>
</tr>
<tr>
<td>MOV EAX ← [y]</td>
<td>MOV EBX ← [x]</td>
</tr>
</tbody>
</table>

Forbidden Final State: Proc 0: EAX = 0 ∧ Proc 1: EBX = 0

Test AMD5.

For x86-TSO, this test has the same force as Example 8.8, but using MFENCE instructions to flush the buffers instead of LOCK’d instructions. The tenth AMD test is similar. None of the Intel litmus tests include fence instructions.

In x86-TSO adding MFENCE between every instruction would clearly suffice to regain sequential consistency (though obviously in practice one would insert fewer barriers), in contrast to IWP/x86-CC/AMD3.14.
Semantics of Linux Spin Locks

On entry the address of spinlock is in register EAX and the spinlock is unlocked iff its value is 1

| acquire: LOCK;DEC [EAX]       | JNS enter                      | ; LOCK’d decrement of [EAX] |
| spin: CMP [EAX],0             | JLE spin                      | ; test [EAX]                |
|                              | JMP acquire                   | ; branch if [EAX] was ≤ 0   |
| enter: ; the critical section starts here |                             | ; try again

| release: MOV [EAX]←1         |

- Question about Linux spin locks: is it OK to have the MOV in release as an unlocked operation?
  - Lets releasing thread continue without flushing write buffer
- Answer: YES!
  - By TSO, the stores within the critical section will all drain from the store buffer before the write that releases the spinlock

Figure credit: [1]


Impact of MM on Applications?

- Most application code is oblivious to processor MM
- Expectation is that code is *data race free (DRF)*
  - data race: $\geq 2$ accesses to the same variable; at least 1 is a write
    - race if two “competing” accesses can execute back-to-back
  - all access to shared memory protected by locks or identified as synchronization accesses if they can race
- Most memory models designed so that race free code behaves *as if* executing on sequentially consistent machine
  - x86-TSO supports interleaving semantics for DRF programs
- Notes
  - spinlock has a race according to above race definition
  - Owens [ECOOP 10] defines “triangular race freedom” to describe this case
    - race between a read and a write where the read is preceded by another write by the same thread and there is no sync operation
    - any x86 TRF program exhibits sequentially consistent semantics
• In sequentially consistent or TSO setting, atomic RMW operations have well-understood semantics
  —e.g., compare-and-swap, load-linked/store-conditional

• In the relaxed setting of Power, what can a programmer depend upon?

• Relaxed memory behavior has never been clearly explained

• Confusion led to Linux kernel bug in implementation of atomic operations
HW Contribution of Synchronizing C/C++ and Power

• Usable model for synchronization primitives, load-linked/store-conditional, on highly relaxed Power architecture

• Define novel abstraction of a write *reaching coherence point* — removes the need for modeling implementation detail of reservation registers for load-linked/store-conditional
Load-Linked (LL) / Store-Conditional (SC)

- RISC-architecture alternative to compare-and-swap
- A LL does a load from some memory address
- A subsequent SC to the same address will succeed or fail
- It will definitely fail if another thread has written to the address in the meantime
- Otherwise, it might succeed, performing a store
- SC sets a flag so that later instructions can determine whether it succeeded or not
- LL/SC pairs often repeated until success

```assembly
1: lwarx r0,0,r2  \ load-reserve from [r2] to r0
    add   r0,r1,r0  \ add register r1 to register r0
    stwcx. r0,0,r2  \ store-conditional r0 to [r2]
    bne-   1b       \ ...looping until success
```
Questions about LL/SC

• Power is highly relaxed and not sequentially consistent

• What does it mean to say “another thread has written to the address between the LL and SC”? 

• What does the Power specification say?
  — informally describes behavior of LL/SC in terms of reservations
  — a SC may succeed only if “the storage location specified by the LL that has established the reservation has not been stored into by another processor or mechanism since the reservation was created” 

• What’s the problem?
  — too much and too little HW detail at the same time
  — the “since” can’t refer to a sequentially consistent order, but the machine execution order is not properly specified
Coherence Order

- For each address, maintain a strict partial order, the **coherence order**, of the ordering commitments that have been made among writes to that address.

- Consider 5 writes by different threads to x.

- Consider the case where writes x=0 and x=2 both propagate to some thread T that didn’t perform either write.
  - T must see x=2, because it is the most recent of the two in the partial order.

Figure credit: [2]
• Introduce to the model of LL/SC the notion of a write reaching coherence point, after which its coherence predecessors are linearly ordered and fixed
  —no additional write can become coherence-before it
Accept a successful write-conditional request A write-conditional request \( w \) by a thread \( tid \), with an accompanying \( wprev \) that was read by the program-order-previous read-reserve, can be accepted and succeed if:

1. the write \( wprev \) is to the same address as \( w \);
2. \( wprev \) has reached coherence point;
3. no coherence-successor of \( wprev \) by another thread has reached coherence point or has been propagated to thread \( tid \); and
4. all writes by \( tid \) to the same address as \( wprev \) (in particular, all those since \( wprev \) was propagated to \( tid \)) have reached coherence point.
Write Reaches Coherence Point

• Write reaches its coherence point: internal transition of storage subsystem for a write already seen if
  —write has not yet reached its coherence point
  —all coherence-predecessor writes have reached their coherence points
  —all writes (to any address and by any thread) propagated to the writing thread before a barrier (also by the writing thread) before this write, have reached their coherence points
Implications

• A store-conditional can succeed even if there are outstanding other writes to the same address by different threads (that might have been read from by yet different threads), but that any such other writes become coherence-later than the store-conditional itself.

• Example

—thread can perform an atomic add of 5 by LL X=0 and doing a successful SC X=5, the latter becoming an immediate coherence successor of i and a coherence predecessor of all other writes.
Store-Conditional Failure

• Store-conditional can fail at any time

• LL reservation may be cleared by many events
  — writes to addresses other than the same cache line
  — hypervisor/OS context switches
  — “implementation-specific characteristics of the coherence mechanism which cause the reservation to be lost”
    – e.g., cache conflict evicts the reserved line
Load-Linked Constraint

- A LL may not be satisfied (by reading a value from the storage subsystem) until all program order previous LL and SC are committed
  —models architectural requirement of single reservation register per thread
Thread Semantics

- LL and SC all commit in program order relative to each other
- LL and SC do not impose any special constraint on normal loads and stores to different addresses
  - do not act as a fence of any kind
  - implications
    - loads after a LL or SC pair might be speculated before it
    - if one wants to prevent that, need to add fence instructions
Forwarding Writes in Power

• Can writes be forwarded to a LL or from a SC within a thread on speculative paths before they have committed to the storage subsystem?

• Such forwarding would make mapping of C/C++11 atomics to Power unsound
Take Away Points

• Looser HW memory models improve performance
  —operation latency can be overlapped with other operations

• HW memory models today are loose in many ways
  —operations within a thread may appear out of order
  —operations by different threads may only be partially ordered

• The x86-TSO model provides an understandable model for programming x86 systems
  —better than prior specifications, which were wrong in different ways

• The Power architecture has very weak ordering
  —constraints are few: writes can even be reordered with LL/SC
  —typically, one must enforce desired orderings with fences