EECS 570
Lecture 14
Memory Consistency

Winter 2017
Prof. Thomas Wenisch

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

Announcements

• Programming Assignment 2
  - Waypoint due on Monday, 3/13
  - Submit via Canvas

• Project Milestone 2
  - Due 3/15
  - Meetings 3/16
  - Prepare a brief slide deck in lieu of a written report
  - Submit via Canvas
Readings

For Wednesday:

✓ Daniel J. Sorin, Mark D. Hill, and David A. Wood, A Primer on Memory Consistency and Cache Coherence (Ch. 3 & 4)

For Monday 3/13

✓ Boehm & Adve - Foundations of the C++ Concurrency Model
Memory Consistency
Memory Consistency Model

A memory (consistency) model specifies the order in which memory accesses performed by one thread become visible to other threads in the program.

It is a contract between the hardware of a shared memory multiprocessor and the successive programming abstractions (instruction set architecture, programming languages) built on top of it.

• Loosely, the memory model specifies:
  □ the set of legal values a load operation can return
  □ the set of legal final memory states for a program
Who cares about memory models?

- Programmers want:
  - A framework for writing correct parallel programs
  - Simple reasoning - “principle of least astonishment”
  - The ability to express as much concurrency as possible

- Compiler/Language designers want:
  - To allow as many compiler optimizations as possible
  - To allow as much implementation flexibility as possible
  - To leave the behavior of “bad” programs undefined

- Hardware/System designers want:
  - To allow as many HW optimizations as possible
  - To minimize hardware requirements / overhead
  - Implementation simplicity (for verification)

We will consider all three perspectives
Uniprocessor memory model

- Loads return value of nearest preceding matched store in <p
  - Need to make partial overlaps work
  - Probably need some special cases for I/O
  - Otherwise, any sort of reordering goes!

- Programmer’s perspective:
  - Generally, no way to tell what order things actually happened

- Compiler’s perspective
  - “as-if” rule – any optimization is legal as long as it produces
    the same output “as-if” executed in program order
  - No “random” changes to memory values (except volatile)

- HW perspective
  - Out-of-order, store buffers, speculation all ok
  - Order only needed per-address, enforced via LSQ
Language-Level
DRF-O Vs SC
Memory Model
Program Order

A ; B

a thread

Execute A and then B
Atomic Shared Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xDEADBEE0</td>
<td>0x0000002a</td>
</tr>
<tr>
<td>0xDEADBEE4</td>
<td>0x00000042</td>
</tr>
<tr>
<td>0xDEADBEE8</td>
<td>0xDEADBEEF</td>
</tr>
</tbody>
</table>

Memory is a map from address to values with reads/writes taking effect immediately.
Intuitive Concurrency Semantics

Shared Memory

Atomic

Program order

Threads

Memory model that guarantees this is called **sequential consistency**
Sequential Consistency

```c
X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();

// Consumer Thread
C: while(!flag);

B: flag = true;
D: x->f++;
```

**sequential consistency (SC)**
[Lamport 1979]
memory operations appear to occur in some global order consistent with the program order
Intuitive reasoning fails in C++/Java

```c
X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;
```

In C++ model this can crash!
Intuitive reasoning fails in C++/Java

X* x = null;
bool flag = false;

// Producer                     // Consumer
A: x = new X();                C: while(!flag);
B: flag = true;                D: x->f++;

B doesn’t depend on A.
It might be faster to reorder them!
Why are accesses reordered?

- **Weak Semantics**: Sequentially valid
- **Performance Optimization**
  - Data-Race-Free-0 Model
    - Java Memory Model
    - C++ Memory Model
  - Buffers
A program has a **data race** if it has an execution in which two **conflicting accesses** to memory are simultaneously ready to execute.

```cpp
// Thread t
A: x = new Data();
C: while(!flag);
B: flag = true;
D: x->f++;
```

Access the same memory location at least one is a write.
Useful Data Races

• Data races are essential for implementing shared-memory synchronization

```
AcquireLock()
{
    while (lock == 1) {}
    t = CAS (lock, 0, 1);
    if (!t) retry;
}

ReleaseLock() {
    lock = 0;
}
```
A program is **data-race-free** if all data races are appropriately annotated (volatile/atomic).

**DRFO**

[Adve & Hill 1990]

SC behavior for data-race-free programs,
weak or no semantics otherwise

**Java Memory Model (JMM)**

[Manson et al. 2005]

**C++0x Memory Model**

[Boehm & Adve 2008]
DRF0-compliant Program

\[
X* \ x = \text{null};
\]
\[
\text{atomic} \ \text{bool} \ \text{flag} = \text{false};
\]

\[
A: \ x = \text{new} \ X();
\]
\[
B: \ \text{flag} = \text{true};
\]
\[
C: \ \text{while}(\text{!flag});
\]
\[
D: \ x->f++;\]

- DRF0 guarantees SC
  .... only if data-race-free (all \textit{unsafe} accesses are annotated)
- What if there is one data-race?
  .... all bets are off (e.g., compiler can output an empty binary!)
Data-Races are Common

• Unintentional data-races
  - Easy to accidentally introduce a data race
    - forget to grab a lock
    - grab the wrong lock
    - forget an atomic annotation
    - ...

• Intentional data-races
  - 100s of “benign” data-races in legacy code
Data Races with no Race Condition (assuming SC)

- Single writer multiple readers

```
// Thread t
A:  time++;

// Thread u
B:  l = time;
```
Data Races with no Race Condition (assuming SC)

• Lazy initialization

// Thread t
if ( p == 0 )
  p = init();

// Thread u
if ( p == 0 )
  p = init();
Intentional Data Races

• ~97% of data races are not errors under SC
  - Experience from one Microsoft internal data-race detection study [Narayanasamy et al. PLDI’07]

• The main reason to annotate data races is to protect against compiler/hardware optimizations
Data Race Detection is Not a Solution

• Current static data-race detectors are not sound and precise
  - typically only handle locks, conservative due to aliasing, ...

• Dynamic analysis is costly
  - DRFx: throw exception on a data-race [Marino’10]
  - Either slow (8x) or requires complex hardware

• Legacy issues
Deficiencies of DRFO

- Weak or no semantics for data-racy programs
- No easy way to identify & reject racy programs

Problematic for DEBUGGABILITY

Analogous to unsafe languages: relying on programmer infallibility

Optimization can lead to jumps to arbitrary code! [Boehm et al., PLDI 2008]

Compiler correctness

Java must maintain safety at the cost of complexity

[Ševčík & Aspinall, ECOOP 2008]
Languages, compilers, processors are adopting DRF0

Not a strong foundation to build our future systems
Language-level SC:
A Safety-First Approach

Program order and shared memory are important abstractions.

Modern languages should protect them.

All programs, buggy or otherwise, should have SC semantics.
What is the Cost of SC?

SC prevents essentially all compiler and hardware optimizations.

And thus SC is impractical.
Sequential Consistency
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
Caches make things more mystifying

• Intuition says: P3 prints A=1
  □ But, with caches:
    ▢ A=0 initially cached at P3 in shared state
    ▢ Invalidation for A arrives at P2; sends out B=1
    ▢ Invalidation for B arrives at P3
    ▢ P3 prints A=0 before invalidation from P1 arrives

• Many past commercial systems allow this behavior
  □ Key issue here: store atomicity
    ▢ Do new values reach all nodes at the same time?
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
  - $A \rightarrow B \land B \rightarrow C \Rightarrow A \rightarrow C$ (transitive)
  - $A \rightarrow A$ (reflexive)
  - $A \rightarrow B \land B \rightarrow A \Rightarrow A = B$ (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 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.
- Switch randomly set after each memory op 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

/* 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>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0</td>
<td>B=0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A=1</td>
<td>B=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld B -&gt; r1;</td>
<td>Ld A -&gt; r1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ld A -&gt; r2;</td>
<td>Ld B -&gt; r2;</td>
<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!)
SC: Programmer’s Perspective

• Generally the least astonishing alternative
  ○ Looks a lot like a multitasking uniprocessor
  ○ Memory behaves as intuition would suggest
  ○ Causality is maintained (SC implies store atomicity)

• But, still plenty of rope to hang yourself
  ○ Any memory access is potentially a synchronization
  ○ Arbitrary wild memory races are legal
  ○ There is still weirdness with C/C++ bit fields
  ○ ...thus, PL still exploring alternative paradigms (e.g., TM)

• And its probably overkill
  ○ Most programmers use libraries for sync...
  ○ ...hence, they don’t actually need SC guarantees
**SC: Compiler's Perspective**

- Disaster! Nearly all optimizations initially appear illegal!
  - Anything could be a sync ⇒ no mem ops may be reordered
  - Effectively disallows:
    - Loop invariant code motion
    - Common sub-expression elimination
    - Register allocation
    - ...

- Not quite that bad...
  - C/C++ specify order only across *sequence points* (statements)
    - Operations within an expression may be reordered

- **Conflict analysis** can improve things [Shasa & Snir’88]
  - Static analysis identifies *conflicting* (racing) accesses
  - Can determine the minimal set of delays to enforce SC
  - But, needs perfect whole-program analysis
Fixing SC Performance

• Option 1: Change the memory model
  - Weak/Relaxed Consistency
  - Programmer specifies when order matters
    - Other access happen concurrently/out-of-order
  + Simple hardware can yield high performance
  - Programmer must reason under counter-intuitive rules

• Option 2: Speculatively ignore ordering rules
  - In-window Speculation & InvisiFence
  - Order matters only if re-orderings are observed
    - Ignore the rules and hope no-one notices
    - Works because data races are rare
  + Performance of relaxed consistency with simple programming model
  - More sophisticated HW; speculation can lead to pathological behavior

One of the most esoteric (but important) topics in multiprocessors
We will study it in-depth after winter break