EECS 570

Lecture 15

Sequential Consistency

Winter 2018

Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eeecs570

Announcements

• Programming Assignment 2
  ❖ Waypoint due on Monday, 3/12
  ❖ Submit via Canvas

• Project Milestone 2
  ❖ Due 3/20
  ❖ Meetings 3/22 and 3/23
  ❖ Prepare a brief slide deck in lieu of a written report
  ❖ Submit via Canvas
Readings

For today:

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

For Wednesday 3/14 (no additional reading)

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

Threads
Intuitive Concurrency Semantics

Shared Memory

Threads

Program order

Instructions in each thread execute in the order they appear in program text
Intuitive Concurrency Semantics

Shared Memory

Threads

Atomic Shared Memory

Memory behaves as if each operation takes effect immediately
Intuitive Concurrency Semantics

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

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

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

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

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

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

Optimizing Compiler and Hardware

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

sequentially valid

Data-Race-Free-0 Model

- Java Memory Model
- C++ Memory Model

buffers
What is DRF-O?

• **Requirement**: Program is data-race-free
  – Programmer distinguishes synchronization variables
    • “volatile” in Java; “atomic” in C++
  – Conflicting data accesses ordered by sync. accesses

• **Guarantee**: Sequential consistency

• **Optimizations** (compiler/hardware)
  – Can freely reorder data accesses between two sync accesses
  – But, ensure SC for sync. accesses (i.e., unsafe racy accesses).
DRFo-compliant Program

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

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

- DRF0 guarantees SC
  .... only if data-race-free (all `unsafe` accesses are annotated)

- What if there is one data-race?
  .... all bets are off (e.g., compiler can output an empty binary!)
DRF0 and Safety

• DRF0 guarantees SC only if data-race-free

• Should we rely on programmer infallibility?
  – Most non-trivial programs will contain a data-race
  – 100s of “benign” data-races in legacy code

• Security issues
  – Just injecting a load might be enough to launch an exploit

• Debugging a racy program will be a nightmare
DRFo and Safety

Languages, compilers, processors in the process of adopting DRFo
-- Not a strong foundation to build our future systems

Need a safety-first approach
Safety-First Proposal

- Assume all accesses are *unsafe (racy)* by default
  - SC for all programs!
  - Can’t reorder any accesses?

Well, that won’t work.

Too slow to be practical?
Safety-First Proposal

• Assume all accesses are unsafe by default
  ⇒ SC for all programs!
  ⇒ Can’t reorder any accesses?

• Find provably safe (non-racy) accesses
  – E.g., Private or read-only shared accesses are safe
  – Static analysis can find them
  – Experts can be allowed to annotate them

• Allow SC-preserving optimizations
  – Most optimizations on safe accesses and several optimizations on unsafe accesses
END-TO-END SEQUENTIAL CONSISTENCY

- Efficient end-to-end SC is feasible with hardware-software cooperation
- SC-Preserving compiler [Marino et al., PLDI ’11]
- SC-Preserving hardware [Singh et al., ISCA’12]
SC HARDWARE
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

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

P1                  P2
A = 1;              B = 1;
if (B != 0) goto retry;   if (A != 0) goto retry;
/* enter critical section*/  /* 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
**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
      
      \[
      \begin{align*}
      P_1 & : A = 0, B = 0, A = 1; \\
      P_2 & : A = 0, B = 1; \\
      P_3 & : Ld B \rightarrow r_1; Ld A \rightarrow r_1; Ld A \rightarrow r_2; Ld B \rightarrow r_2;
      \end{align*}
      \]

  - 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!)
Memory operations in pipeline can not 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]
Memory Access Classification

- Two complementary access classification schemes
  - Static compiler analysis
  - Dynamic page protection mechanism
Unsafe loads must wait for unsafe store buffer drain

Old Rules
Conventional SC

Unsafe stores must retire in-order

New Rules
Access Type Aware SC

Unsafe loads must wait for unsafe store buffer drain

Unsafe stores must retire in-order
Cannot commit a load when a store is still pending.

Commit unsafe loads even when safe stores are pending.

Commit safe loads always.

Coalesce & retire out-of-order.
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
SIMULATION METHODOLOGY

- LLVM compiler extensions
  - SC-preserving
  - Support for static classification

- Hardware simulator
  - Simics based FeS2, x86_64

- Benchmarks:
  - PARSEC, SPLASH-2, Apache web server (SURGE)

- Compare End-To-End SC to
  - Stock LLVM on TSO
  - Stock LLVM on RMO
# Hardware Configuration

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>16 Cores @ 4Ghz</td>
</tr>
<tr>
<td>Fetch/Exec/ Commit width</td>
<td>4 instructions per core</td>
</tr>
</tbody>
</table>
| Store Buffer     | *Unsafe* Store buffer:  
|                  | 64 8-byte entry FIFO buffer  
|                  | *Safe* Store Buffer:  
|                  | 8 64-byte entry coalescing un-ordered buffer |
| L1 Cache         | 64 KB per-core (private) |
| L2 Cache         | 1MB private            |
| Coherence        | MOESI directory        |
| Memory           | 160 cycles             |
Average performance cost of end-to-end SC is 6.2% w.r.t stock compiler on TSO
**End-to-End Sequential Consistency**

- **DRF0**: assume a memory access is *safe* by default
- **SC**: assumes a memory access is *unsafe* by default

**SC-Preserving compiler**
- Optimizations that break SC don’t buy much performance
- Exposing hardware load speculation enables more optimizations

**SC Hardware**
- Identify safe accesses using compiler and OS
- Relax memory ordering constraints for safe accesses

**Overhead over stock**: avg. ~6%
SC-PRESERVING COMPILER
SC-Preserving Definition

• A SC-preserving compiler ensures that every SC-behavior of the binary is a SC-behavior of the source

• Guarantees end-to-end SC when the binary runs on SC HW
An SC-Preserving C Compiler

modified LLVM [Lattner & Adve 2004] to be SC-preserving

- obvious idea: restrict optimizations so they never reorder shared accesses
- simple, small modifications to the base compiler
- slowdown on x86: average of 3.8%
  - PARSEC, SPLASH-2, SPEC CINT2006
Some optimizations preserve SC

- loop unrolling
- arithmetic reassociation
- stack slot coloring
- unreachable code elim.
- dead argument elim.
- loop rotation
- loop unswitching
- correlated val prop
- allocating locals to virtual registers
- scalar replication
- tail call elim

virtual to physical register allocation

arithmetc simplification

for (i=0; i<3; i++)
    X++;

loop unrolling

X++; X++; X++

foo();
bar();
baz();

function inlining

foo();

X++;

bar() {X++;}

baz();

unreachable code elim.

dead argument elim.

stack slot coloring

virtual to physical register allocation

t=X*4;

arithmetic simplification

virtual to physical register allocation

t=X<<2;

dead argument elim.

loop rotation

loop unswitching

correlated val prop

allocating locals to virtual registers

Many
Optimizations That Break SC

• Example: Common Subexpression Elimination (CSE)

L1: \( t = X \times 5; \)
L2: \( u = Y; \)
L3: \( v = X \times 5; \)

L1: \( t = X \times 5; \)
L2: \( u = Y; \)
L3: \( v = t; \)

\( t,u,v \) are local variables
\( X,Y \) are possibly shared
Common Subexpression Elimination is not SC-Preserving

Init: $X = Y = 0$

Left:

- L1: $t = X \times 5$
- L2: $u = Y$
- L3: $v = X \times 5$

M1: $X = 1$
M2: $Y = 1$

$u = 1 \implies v = 5$

Right:

- L1: $t = X \times 5$
- L2: $u = Y$
- L3: $v = t$

M1: $X = 1$
M2: $Y = 1$

Possibly $u = 1 \land v = 0$
Implementing CSE in a SC-Preserving Compiler

- Enable this transformation when
  - X is a *safe* variable, or
  - Y is a *safe* variable

- Identifying *safe* variables:
  - Compiler generated temporaries
  - Stack allocated variables whose address is not taken

- More *safe* variables?
A SC-preserved LLVM for C programs

• Enable transformations on *safe* variables
• Enable transformations involving a single shared variable
  ❑ e.g. \( t = X; u = X; v = X; \rightarrow t = X; u = t; v = t; \)
• Enable trace-preserving optimizations
  ❑ These do not change the order of memory operations
  ❑ e.g. loop unrolling, procedure inlining, control-flow simplification, dead-code elimination,…
• Modified each of ~70 passes in LLVM to be SC-preserved
Experiments using LLVM

• baseline
  stock LLVM compiler with standard optimizations (-O3)
• no optimizations
  disable all LLVM optimization passes
• naïve SC-preserving
  disable LLVM passes that possibly reorder memory accesses
• SC-preserving
  use modified LLVM passes that avoid reordering shared memory accesses

• ran compiled programs on 8-core Intel Xeon
Parallel Benchmarks

Slowdown over LLVM –O3

No opts.

Naïve

SC-preserving

SC-preserving
SPEC Integer 2006

Slowdown over LLVM –O3

No optimization  Naïve SC-preserving  SC-preserving

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>No optimization</th>
<th>Naïve SC-preserving</th>
<th>SC-preserving</th>
</tr>
</thead>
<tbody>
<tr>
<td>400.perlbench</td>
<td>149</td>
<td></td>
<td></td>
</tr>
<tr>
<td>401.bzip2</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>403 gcc</td>
<td>149</td>
<td></td>
<td></td>
</tr>
<tr>
<td>429.mcf</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>445.gobmk</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>456.hammer</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>458.sjeng</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>462.libquantum</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>464.h264ref</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>471.omnetpp</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>473.astar</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>483.xalanbench</td>
<td>170</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td>170</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How Far Can SC-Preserving Compiler Go?

float s, *x, *y;
int i;
s=0;
for( i=0; i<n; i++ ){
    s += (x[i]-y[i]) * (x[i]-y[i]);
}

can not be optimized

float s, *x, *y;
int i;
s=0;
for( i=0; i<n; i++ ){
    s += (*x + i*sizeof(float)) - *(y + i*sizeof(float)) * (*x + i*sizeof(float)) - *(y + i*sizeof(float));
}

SC preserving

float s, *x, *y;
float *px, *py, *e;
s=0; py=y; e = &x[n]
for( px=x; px<e; px++, py++){
    s += (*px-*py) * (*px-*py);
}

full optimization

float s, *x, *y;
float *px, *py, *e, t;
s=0; py=y; e = &x[n]
for( px=x; px<e; px++, py++){
    t = (*px-*py);
    s += t*t;
}
Many “Can’t-Live-Without” Optimizations are Eager-Load Optimizations

- Eagerly perform loads or use values from previous loads or stores

**Common Subexpression Elimination**

L1: \( t = X \times 5; \)
L2: \( u = Y; \)
L3: \( v = X \times 5; \)

L1: \( t = X \times 5; \)
L2: \( u = Y; \)
L3: \( v = t; \)

**Constant/copy Propagation**

L1: \( X = 2; \)
L2: \( u = Y; \)
L3: \( v = X \times 5; \)

L1: \( X = 2; \)
L2: \( u = Y; \)
L3: \( v = 10; \)

**Loop-invariant Code Motion**

L1:
L2: for(…)
L3: \( t = X \times 5; \)

L1: \( u = X \times 5; \)
L2: for(…)
L3: \( t = u; \)
Performance overhead

Allowing eager-load optimizations alone reduces max overhead to 6%
Speculatively Performing Eager-Load Optimizations

- On `monitor.load`, hardware starts tracking coherence messages on X’s cache line.
- The interference check fails if X’s cache line has been downgraded since the `monitor.load`.
- In our implementation, a single instruction checks interference on up to 32 tags.

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;

L1: t = monitor.load(X, tag) * 5;
L2: u = Y;
L3: v = t;
C4: if (interference.check(tag))
C5:   v = X*5;
```
Conclusion on SC-Preserving Compiler

- Efficient SC-preserving compiler is feasible with careful engineering
- Hardware support can enable eager-load optimizations without violating SC
Fixing SC Performance

• Option 1: Change the memory model
  □ Weak/Relaxed Consistency
  □ Programmer specifies when order matters
    ○ Other accesses 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