Relaxed Consistency
Announcements

Midterm, Monday 3/15

- Open book.
- All lectures and reading assignments till today.
  - Excluding memory consistency models

- By popular demand:
  - Take-home (24 hr exam)
  - Released at 1:30pm 3/15 on Gradescope
  - Submit before 1:29pm on 3/16 on Gradescope
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-0 Vs SC
Memory Model
Program Order

a thread

A ;
B

Execute A and then B
Atomic Shared Memory

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

Memory model that guarantees this is called **sequential consistency**
X* x = null;
bool flag = false;

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

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

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

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

Programming Language

- Compiler
  - sequentially valid optimizations can reorder memory accesses
    - e.g. common subexpression elimination, register promotion, instruction scheduling

Hardware

- sequentially valid optimizations can reorder memory accesses
  - e.g. out-of-order execution, store buffers
  - performance optimization

weak semantics

- Data-Race-Free-0 Model
  - Java Memory Model
  - C++ Memory Model
  - e.g. Data-Race-Free-0, weak semantics
A program has a data race if it has an execution in which two conflicting accesses to memory are simultaneously ready to execute.
Useful Data Races

- Data races are essential for implementing shared-memory synchronization

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

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

**DRF0**
[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

```cpp
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!)
Data-Races are Common

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

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

- Single writer multiple readers

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

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

• Lazy initialization

```c
// 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
  r 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 \textit{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 DRF0

- 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 + data race = jump to arbitrary code! [Boehm et al., PLDI 2008]

Correctness with 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  Processor 1
A=1;       while (!flag); // spin
flag=1;    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

A=0  B=0

P1  P2                  P3
A=1; while (A==0); while (B==0);  
B = 1;  print A;

- **Intuition says**: P3 prints A=1
  - But, with caches:
    - A=0 initially cached at P3 in shared state
    - Update for A arrives at P2; sends out B=1
    - Update for B arrives at P3
    - P3 prints A=0 before update 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

- \( r \) Guarantees a total order for all accesses to a location that is consistent with the program order
  - \( m \) 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
  \[ \text{Program order} (\!< p \!) \text{ – per-thread order in inst. sequence} \]
  \[ \text{Memory order} (\!< M \!) \text{ – order memory ops are performed} \]
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:
  m Before LOAD is performed w.r.t. any other processor, all prior LOADs must be globally performed and all prior STOREs must be performed
  m Before STORE is performed w.r.t. any other processor, all prior LOADs globally performed and all previous STORE be performed.
  m 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

P1 P2 P3

switch randomly set after each memory op provides total order among all operations

Memory
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        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
Write A
Read B
t1
t3

P2
Read A
Write B
t2
t4

Shared Bus

A: 0
B: 0

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

\[ m \] Processor core waits for store to complete, before issuing next memory op

\[ m \] 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
      
      \[
      \begin{align*}
      A &= 0 \quad B = 0 \\
      P1: \quad A &= 1; \quad B \quad = \quad 1; \\
      P2: \quad \text{Ld } B \quad \rightarrow \quad r1; \quad \text{Ld } A \quad \rightarrow \quad r1; \\
      P3: \quad \text{Ld } A \quad \rightarrow \quad r2; \quad \text{Ld } B \quad \rightarrow \quad r2; \\
      \end{align*}
      \]
  
  - Under store-atomicity, what results are (im-)possible?
Implementing Store Atomicity

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

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

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

• With multiprocessors & shared caches
  r 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!
  r Anything could be a sync ⇒ no mem ops may be reordered
  r Effectively disallows:
    m Loop invariant code motion
    m Common sub-expression elimination
    m Register allocation
    m ...

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

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

• Option 1: Change the memory model
  r Weak/Relaxed Consistency
  r Programmer specifies when order matters
    m 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
  r In-window Speculation & InvisiFence
  r Order matters only if re-orderings are observed
    m Ignore the rules and hope no-one notices
    m 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
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

**all optimizations** on locals and compiler temporaries

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

loop unrolling

X++; X++; X++

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

function inlining

foo();
X++;
bar(){X++;}
baz();

arithmetic reassociation

stack slot coloring

t=X*4;

arithmetic simplification

t=X<<2;

unreachable code elim.

dead argument elim.

loop rotation

loop unswitching

virtual to physical register allocation

scalar replication

tail call elim

allocating locals to virtual registers

correlated val prop
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: \(X = Y = 0;\)

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

M1: \(X = 1;\)
M2: \(Y = 1;\)

possibly \(u = 1 \land v = 0\)

Init: \(X = Y = 0;\)

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

M1: \(X = 1;\)
M2: \(Y = 1;\)

\(u = 1 \land v = 5\)
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-preserving LLVM for C programs

- Enable transformations on *safe* variables
- Enable transformations involving a single shared variable
  - e.g. \( t = X; u = X; v = X; \) \( t = X; u = t; v = t; \)
- 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-preserving
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
How Far Can SC-Preserving Compiler Go?

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

```c
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)));
}
```

```c
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);
}
```

```c
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

\begin{align*}
L1: & \ t = X*5; \\
L2: & \ u = Y; \\
L3: & \ v = X*5;
\end{align*}

\begin{align*}
L1: & \ t = X*5; \\
L2: & \ u = Y; \\
L3: & \ v = t;
\end{align*}

Constant/copy Propagation

\begin{align*}
L1: & \ X = 2; \\
L2: & \ u = Y; \\
L3: & \ v = X*5;
\end{align*}

\begin{align*}
L1: & \ X = 2; \\
L2: & \ u = Y; \\
L3: & \ v = 10;
\end{align*}

Loop-invariant Code Motion

\begin{align*}
L1: & \ \\
L2: & \ for(...) \\
L3: & \ t = X*5;
\end{align*}

\begin{align*}
L1: & \ u = X*5; \\
L2: & \ for(...) \\
L3: & \ t = u;
\end{align*}
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.
Conclusion on SC-Preserving Compiler

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