Announcements

• Programming Assignment 2
  ✓ Waypoint due on Monday, 3/18
  ✓ Submit via Canvas

• Project Milestone 2
  ✓ Due 3/27
  ✓ Meetings 3/29
  ✓ Prepare a brief slide deck in lieu of a written report
  ✓ Submit via Canvas
Readings

For Today:


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

For Monday 3/18

- Alglave et al - Frightening Small Children and Disconcerting Adults: Concurrency in the Linux Kernel
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

  \[
  \begin{array}{l}
  \text{P1} \\
  \text{A=0 B=0} \\
  \text{A=1;}
  \\
  \text{P2} \\
  \text{B = 1;}
  \\
  \text{P3} \\
  \text{Ld B \rightarrow r1;}
  \\
  \text{Ld A \rightarrow r2;}
  \\
  \text{P4} \\
  \text{Ld A \rightarrow r1;}
  \\
  \text{Ld B \rightarrow r2;}
  \end{array}
  \]

  □ 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 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
Many optimizations preserve SC

**all optimizations** on locals and compiler temporaries

```c
for(i=0; i<3; i++)
    X++;
loop unrolling
X++; X++; X++

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

arithmetic reassociation
foo();
X++;
baz();

stack slot coloring

unreachable code elim.
t=X*4;
arithmetic simplification
t=X<<2;
dead argument elim.

virtual to physical register allocation
loop rotation
loop unswitching

scalar replication
correlated val prop
allocating locals to virtual registers
tail call elim
Optimizations That Break SC

• Example: Common Subexpression Elimination (CSE)

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

\[\rightarrow\]

```
L1: t = X*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: \( t = X*5; \) | M1: \( X = 1; \) |
| L2: \( u = Y; \) | M2: \( Y = 1; \) |
| L3: \( v = X*5; \) | L1: \( t = X*5; \) |
| M1: \( X = 1; \) | M2: \( Y = 1; \) |
| M2: \( Y = 1; \) | L3: \( v = t; \) |

\[ u == 1 \implies v == 5 \]

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

| L1: \( t = X*5; \) | M1: \( X = 1; \) |
| L2: \( u = Y; \) | M2: \( Y = 1; \) |
| L3: \( v = t; \) | M1: \( X = 1; \) |
| M2: \( Y = 1; \) |

Possibly \( u == 1 \) \&\& \( v == 0 \)
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-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;  \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-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

- No optimization
- Naïve SC-preserving
- SC-preserving

Slowdown percentage for various benchmarks over LLVM with -O3 optimization.
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]);
}

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);
}

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: \( v = X \times 5; \)

\[ \rightarrow \]

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

**Constant/copy Propagation**

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

\[ \rightarrow \]

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

**Loop-invariant Code Motion**

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

\[ \rightarrow \]

L1: 
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:  | `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
Relaxed Consistency
Review: Problems with SC

• 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
Execution in strict SC

- Miss on Wr A stalls all unrelated accesses

CPU pipeline

<table>
<thead>
<tr>
<th>Wr A</th>
<th>Rd B</th>
<th>Rd C</th>
<th>Wr D</th>
<th>Rd E</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Wr A</th>
<th>Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd B</td>
<td>Idle</td>
</tr>
<tr>
<td>Rd C</td>
<td>Idle</td>
</tr>
<tr>
<td>Wr D</td>
<td>Idle</td>
</tr>
</tbody>
</table>

Memory accesses issue one-at-a-time
Relaxing Write-to-Read Order

- Motivation: Post-retirement store buffers

- Allow reads to bypass incomplete writes
  - Reads search store buffer for matching values
  - Hides all latency of store misses in uniprocessors

- Writes are still ordered w.r.t. other writes

- Reads are still ordered w.r.t. other reads
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*/
Processor Consistency (PC)

• Formal requirements [Goodman 1989]:
  □ Before LOAD is performed w.r.t. any other processor, all prior LOADs must be performed
  □ Before STORE is performed w.r.t. any other processor, all prior mem ops must be performed

• Does not require store atomicity
• This is basically what x86 and VAX do
• Also roughly what IBM’s System-370 did (1960’s)
  □ Oddly, on 370, loads must stall if they hit in the SB
Sun’s “Total Store Order” (TSO)

• Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is *total* for all operations with store semantics
  - **Atomicity** - Atomic read-modify-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered

• This is the same as PC, except that it requires store atomicity
PC/TSO: Programmer's Perspective

- Can occasionally lead to astonishing behavior changes
  - E.g., Dekker’s algorithm doesn’t work
  - ISAs provide an STBAR (store barrier) to manually force order
    - Semantics – store buffer must be empty before memory operations after STBAR may be executed
  - Can also enforce order by replacing stores with RMWs
- But, the key case, where sync is done with locks, simply works
  - Lock acquires are RMW operations ⇒ they force order for preceding/succeeding loads/stores
    - Load semantics of RMW imply load-load orderings
    - Ditto for store semantics
  - Lock release is a store operation ⇒ it must be performed after critical section is done
PC/TSO: Compiler’s Perspective

- Compiler may now hoist loads across stores
  - Still can’t reorder stores or move loads across loads
  - Not clear how helpful this is in practice...
  - ...Recent results from Prof. Satish’s group:
    - 5-30% perf. gap vs. compiler that preserves SC [PLDI’11]

- No new crazy memory barriers to emit
  - Library/OS writers need to use a little bit of caution;
    use RMWs instead of loads/stores for synchronization
  - TSO-unsafe code is rare enough that it can be the programmer’s problem

- No need to invoke “undefined behavior” to avoid onerous implementation requirements
PC/TSO: HW Perspective

- Allows a FIFO-ordered, non-coalescing store buffer
  - Typically maintains stores at word-granularity
  - Loads search buffer for matching store(s)
    - Some ISAs must deal with merging partial load matches
  - Coalescing only allowed among adjacent stores to same block
  - Must force buffer to drain on RMW and STBAR
  - Often, this is implemented in same HW structure as (speculative) store queue

- Can hide store latency!
  - But, store buffer may need to be quite big
    - Stores that will be cache hits remain buffered behind misses
  - Associative search limits scalability
    - E.g., certainly no more than 64 entries
Execution in PC / TSO

- Stores misses do not stall retirement
  - St A, St C, Ld D misses overlapped
  - Ld B retired without waiting for St A to fill
  - St C consumes space in store buffer even though it will hit

PC/TSO hides store miss latency
Relaxing Write-to-Write Order

- Motivation: Coalescing store buffers & early drain

- Allows writes to coalesce in SB & drain early

```
CPU pipeline

retired
St A
Ld B
St C
Ld D
Ld E

St A Miss
Ld B Retired
St C Hit
Ld D Miss
Ld E Idle

Allows us to drain this
```
Sun’s “Partial Store Order” (PSO)

- Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is total for all operations with store semantics
  - **Atomicity** - Atomic read-modify-write-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered only if they are separated by a *membar* (memory barrier) instruction
  - **StoreStoreEq** – Stores to the same address are ordered
PSO: Compiler/HW Perspective

- Allows an unordered, coalescing post-retirement store buffer
  - Can now use a cache-block-grain set-associative structure
  - Store misses leave store buffer upon cache fill
  - Loads search buffer for matching store(s)
  - Must force buffer to drain on RMW and STBAR

- Much more efficient store buffer

- But, still doesn’t allow out-of-order loads
  - No OoO execution (without speculation)
  - Compiler’s hands are still tied
Relaxing all Order

- Motivation: Out-of-order execution & multiple load misses

- Now Ld E can complete even though earlier insn aren’t done
Two ISA Approaches to enforce order

• Approach 1: Using explicit “fence” (aka memory barrier)
  □ Sun’s Relaxed Memory Order (RMO), Alpha, PowerPC, ARM
    Ld, St, ...
    L L S S
    Fence      Enforces order if bit is set
    L S L S
    Ld, St, ...

• Approach 2: Annotate loads/stores that do synchronization
  □ Weak Ordering, Release Consistency (RC)
  □ Data-Race-Free-0 (DRF0) – prog. language-level model
    Load.acquire       Lock1
    ...
    Store.release      Lock1
More definitions... Dependence Order

- A refinement of program order ($<p$)

- Dependence order ($<d$) captures the minimal subset of ($<p$) that guarantees self-consistent execution traces. $X <p Y$ implies $X <d Y$ if at least one of the following is true:
  - The execution of $Y$ is conditional on $X$ and $S(Y)$ ($Y$ is a store)
  - $Y$ reads a register that is written by $X$
  - $X$ and $Y$ access the same memory location and $S(X)$ and $L(Y)$

- Dependence order captures what an out-of-order core is allowed to do (ignoring exceptions)
Sun’s “Relaxed Memory Order” (RMO)

- Formal requirements [v9 architecture manual]:
  - $X <d Y \land L(X) \Rightarrow X <M Y$
    - RMO will maintain dependence order if preceding insn. is a load
  - $M(X,Y) \Rightarrow X <M Y$
    - MEMBAR instructions order memory operations
  - $Xa <p Ya \land S(Y) \Rightarrow X <M Y$
    - Stores to the same address are performed in program order
Execution in RMO w/ fence

- "Fence" indicates ordering affects correctness
  - Retirement stalls at Fence
  - Typically, accesses after fence don’t issue
RMO: Programmer’s Perspective

- Programmer must specify MEMBARS wherever they are needed
  - This is hard; Specifying minimum barriers is harder
  - E.g., lock and unlock (from v9 ref. manual; w/o branch delays)

```assembly
LockWithLDSTUB (lock)
  retry:   ldstub [lock],%10
            tst %10
            be out
  loop:    ldub [lock],%10
            tst %10
            bne loop
            ba,a retry
  out:     membar #LoadLoad | #LoadStore

UnLockWithLDSTUB (lock)
  membar #StoreStore !RMO and PSO only
  membar #LoadStore     !RMO only
  stub %g0,[lock]
```
RMO: Compiler’s Perspective

- **Sweet, sweet freedom!**
  - Compiler may freely re-order between fences
  - Also across fences if the fence allows it (partial fences)
  - Note: still can’t reorder stores to same address
  - Programmer’s problem if the fences are wrong...
RMO: HW Perspective

- Unordered, coalescing post-retirement store buffer
  - Just like PSO

- Out-of-order execution!
  - Need a standard uniprocessor store queue
  - Fence instruction implementation
    - Easy – stall at issue
    - Hard – retire to store buffer and track precise constraints
**Weak Ordering (WO)**

- Loads and stores can be labeled as “sync”
  - No reordering allowed across sync instructions
Release Consistency (RC)

- Specialize loads and stores as “acquires” and “releases”
  - Load at top of critical section is an acquire
  - Store at bottom of critical section is a release
  - Allows reordering into critical sections, but not out

![Diagram of release consistency](image)

Weak Consistency (WCsc)

Release Consistency (RCpc)

u v cannot perform with respect to any other processor until u is performed
WO/RC: Programmer Perspective

• A new way of thinking: programmer-centric memory models
  - If you annotate syncs correctly, your program will behave like SC
  - E.g., Data-Race-Free-0 (DRF0)
    - Accesses are either normal or sync
    - Sync accesses are sequentially consistent, and order normal accesses
    - Data races among normal accesses are prohibited
    - DRF0 programs appear as if they ran under SC
  - Lot’s of ongoing work on race detection
WO/RC: Compiler’s Perspective

- DRF0 is foundation of C++ and Java memory models
  - Compiler may freely re-order between syncs
  - But, it may not introduce any new data races
    - Can’t do speculative loads (can disallow optimizations)

- Why are races so evil?
  - They break reasonable optimization:
    ```
    unsigned int i = x; // x is shared variable
    if (i < 2) {
        // opt: don’t copy x, use by ref
        foo: ...
        // suppose x changes here...
        switch (i) {
            // opt: implement as jump table
            case 0: ... break;
            case 1: ... break;
            default: ... break; // opt: range inference tells
                             // compiler this case is
                             // impossible, drop the check
        }
    }
    ```