CS315A/EE382B: Lecture 11

Memory Consistency & CMP Introduction

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

Announcements

- PA2 due Wed May 17
- Midterms back today

Today's Outline

- Memory Consistency
- CMPs

Synchronization Summary

- Interaction of hardware-software tradeoffs
- Must evaluate hardware primitives and software algorithms together
  - primitives determine which algorithms perform well
- Simple software algorithms with common hardware primitives do well on bus

Coherence vs. Consistency

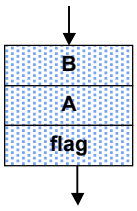
- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations
- A Memory System is Coherent if
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    - program order = order defined program text or assembly code
  - value returned by a read is value written by last store to that location

Why Coherence != Consistency

/* initial A = B = flag = 0 */

P1	P2
`A = 1;`	`while (flag == 0); /* spin */`
`B = 1;`	`print A;`
`flag = 1;`	`print B;`

Intuition says printed A = B = 1
Coherence doesn't say anything, why?
Consider a coalescing write buffer



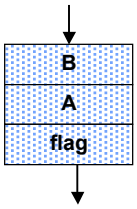
Why Coherence != Consistency

```
/* initial A = B = flag = 0 */
 P1                    P2
A = 1;                while (flag == 0); /* spin */
B = 1;                print A;
flag = 1;             print B;
```

Intuition says printed A = B = 1

Coherence doesn't say anything, why?

Consider a coalescing write buffer



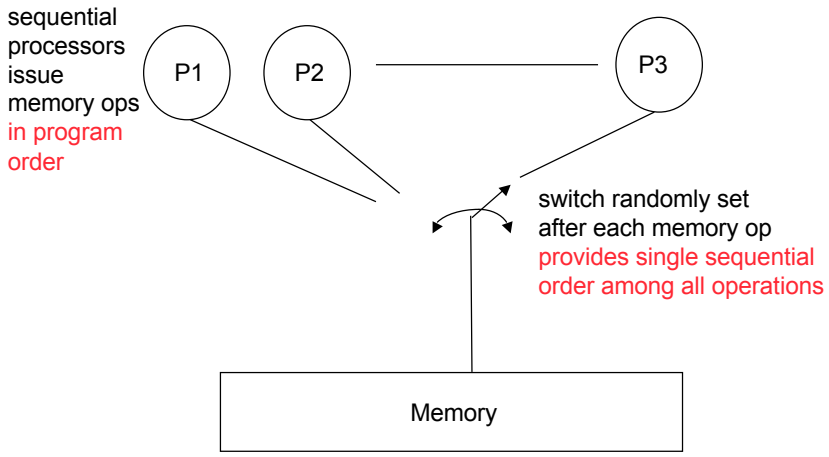
Memory Consistency Model

- Interface between programmer and system
- What levels of the system do you need a memory consistency model?

Sequential Consistency

- Lamport 1979
- 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
  - The operations of each individual processor appear in this sequence in the order specified by its program
- What a moderately sophisticated software person would assume of shared memory

Sequential Consistency (SC)



Definitions and Sufficient Conditions

- Sequentially Consistent Execution
  - result is same as one of the possible interleavings on uniprocessor
- Sequentially Consistent System
  - All execution is sequentially consistent
  - Not possible to get execution that does not correspond to some possible total order

Memory Consistency Definitions

- Memory operation
  - execution of load, store, atomic read-modify-write access to memory location
- Issue
  - operation is issued when it leaves processor and is presented to memory system (cache, write-buffer, local and remote memories)
- Perform
  - A store is performed wrt to a processor p when a load by p returns value produced by that store or a later store
  - A load is performed wrt to a processor when subsequent stores cannot affect value returned by that load
- Complete
  - memory operation is performed wrt all processors.

Sufficient Conditions for Sequential Consistency

- Every processor issues memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced the value to complete before issuing next op
  - Ensures write atomicity (2 conditions)
    - Writes to same location are serialized
    - Can't read result of store until all processors will see new value
- Easily implemented with shared bus

Impact of Sequential Consistency (SC)

- Literal implementation: one memory module and no caches
- High performance implementations
  - Coherent caching
    - How do you maintain write atomicity with invalidates?
  - Non binding prefetch
    - What is this?
  - Multithreading
    - Is write atomicity a problem?
- Compilers
  - No reordering of shared memory operations
    - What simple optimizations does this disallow?
  - Why is register allocation of shared memory bad?
- Why are'nt most modern systems sequentially consistent?

Relaxed Memory Models

- Motivation is increased performance
  - Overlap multiple reads and writes in the memory system
  - Execute reads as early as possible and writes as late as possible

- Rely on "synchronized" programs to get same behavior as SC
  - What is a synchronized program?
- Recall SC has
  - Each processor generates at total order of its reads and writes (R → R, R → W, W → W, & W → R)
  - That are interleaved into a global total order
- (Most) Relaxed Models
- Processor consistency (PC):
  - Relax ordering from writes to (other proc's) reads
- Relaxed consistency (RC):
  - Relax all read/write orderings (but add "fences")

Relax Write to Read Order

```
        /* initial A = B = 0 */
  P1                    P2
A = 1;                B = 1;
r1 = B;               r2 = A;
```

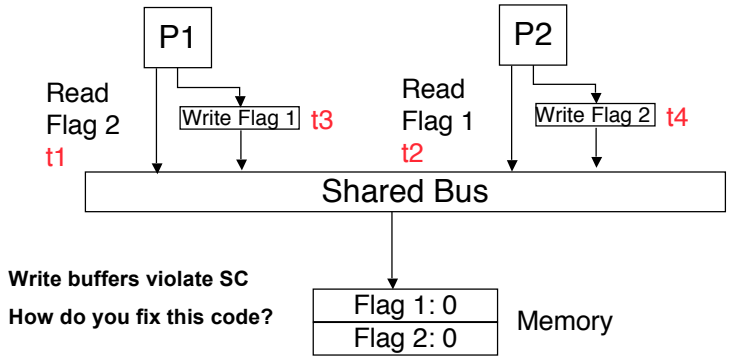
Processor Consistent (PC) Models

Allow r1==r2==0 (precluded by SC, why?)

Examples: IBM 370, Sun's Total Store Order, & Intel IA-32

Why do this?

Write Buffers w/ Read Bypass




```
P1                          P2
Flag 1 = 1                  Flag 2 = 1
if (Flag 2 == 0)            if (Flag 1 == 0)
   critical section            critical section
```


Why Not Relax All Orders?

```
     /* initially all 0 */
 P1                    P2
A = 1;              while (flag == 0); /* spin */
B = 1;              r1 = A;
flag = 1;           r2 = B;
```

Reorder of “A = 1”,“B = 1” or “r1 = A”,“r2 = B”

Via OOO processor, non-FIFO write buffers, delayed invalidation acknowledgements, etc.

But Must Order

“A = 1”,“B = 1” before “flag =1”

“flag != 0” before “r1 = A”,“r2 = B”

Order with "Synch" Operations "Safety Nets"

```
        /* initially all 0 */
 P1                    P2
A = 1;              while (SYNCH  flag == 0);
B = 1;              r1 = A;
SYNCH flag = 1;     r2 = B;
```

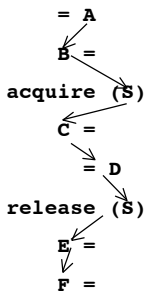
Called "weak ordering" of "weak consistency" (WC)

Alternatively, relaxed consistency (RC) specializes

- Acquires: force subsequent reads/writes after
- Releases: force previous reads/writes before

Sequential Consistency and Processor Consistency

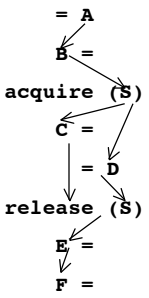
SC



Normal orderings: R→R, R→W, W→R, W→W

Synch. orderings: S→R, S→W, R→S, W→S, S→S

PC/TSO

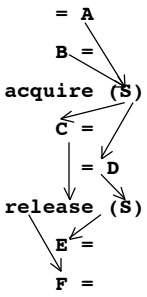


Normal orderings: R→R, R→W, W→W

Synch. orderings: S→R, S→W, R→S, W→S, S→S

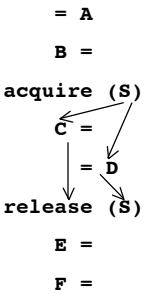
Weak Ordering and Release Consistency

WO



Synch. orderings: S→R, S→W, R→S, W→S, S→S

RC



Synch. orderings: S_A→R, S_A→W, R→S_R, W→S_R, S_A→S_A , S_A→S_r, S_r→S_A, S_R→S_R

Commercial Models use "Fences"

```
/* initially all 0 */
 P1                    P2
A = 1;                while (flag == 0);
B = 1;                FENCE;
FENCE;                r1 = A;
flag = 1;             r2 = B;
```

Examples: Compaq Alpha, IBM PowerPC,& Sun RMO

Can specialize fence: write fences and read fences (e.g., RMO)

The Programming Interface

- WO and RC require synchronized programs
  - All access to shared data separated by a pair of sync. ops.
  - Data-race free

```
write (x)
...
release (s)
...
acquire (s)
...
access (x)
```

- All synchronization operations must be labeled and visible to the hardware
  - easy if synchronization library used
  - must provide language support for arbitrary ld/st synchronization (event notification, e.g., flag)
- Program that is correct for TSO portable to WO & RC

Implementing Relaxed Models

- Processor consistency
  - Read misses bypass pending writes
  - Write-buffer with tag check
  - Memory and bus that supports two outstanding misses
  - Hide latency of write misses

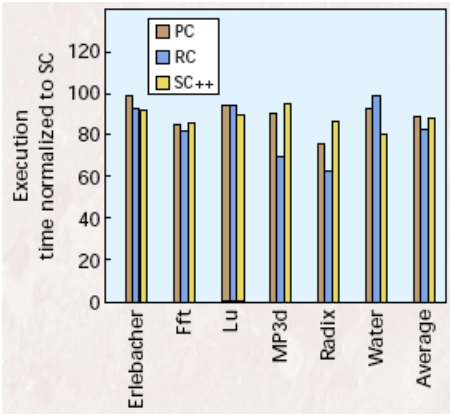
- Release consistency
  - Allow multiple outstanding writes
  - Read misses bypass writes
  - Processor must have nonblocking (lockup free) cache
  - Memory and bus that supports multiple outstanding misses
  - Hide more write latency and read latency

SC and RC Performance Gap

- Relaxed models offer more implementation options
  - Write buffers
  - Nonblocking caches
- Improving SC performance
  - Don't serialize coherence permission operations
    - How do you execute read A, write B, read C, write D?
  - Speculative execution
    - Allows SC implementations to hide read latency
    - How does this work?
    - Read A (miss), read B (hit)
- Closes gap between SC and RC

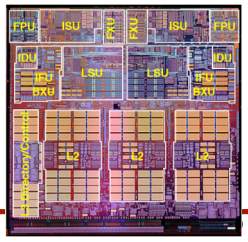
Do We Need Relaxed Models?

- Mark Hill says we don't
- Sequential consistency (SC) is enough
  - processor consistency OK
  - Need speculative execution (SE)
- Speculation already a core part of microprocessor design
- SC+SE within 16% of relaxed models on scientific benchmarks
  - What is impact on commercial apps?
- Makes life simpler for the parallel programmer
- SC restricts compiler optimizations



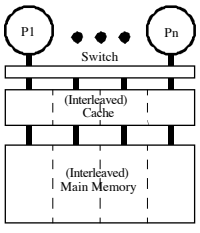
Single Thread Performance has Reached Limits

- This has been said before, but it is really happening now!
- ILP and deep pipelining have run out of steam:
  - ILP parallelism in applications has been mined out
  - Communication delays are hurting complex microarchitectures
  - Frequency scaling is now technology-driven (minimal pipe stages)
  - The power and complexity of microarchitectures taxes our ability to cool and verify
- Latest evidence
  - Intel cancels Tejas: projected 180 W
  - Sun cancels Millennium : 4 years late
  - Comparable performance
- Intel, IBM, Sun pursuing multicore designs



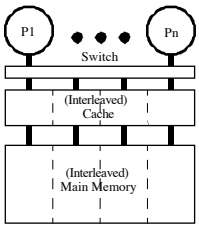
Shared Cache Advantages

- Cache placement identical to single cache
  - Only one copy of any cached block
- Fine-grain sharing
  - Communication latency determined level in the storage hierarchy where the access paths meet
  - 10– 20 cycles for L2 cache meeting
- Potential for positive interference
  - One proc. prefetches data for another
- Smaller total storage
  - Only one copy of code/data used by both proc.
- Can share data within a line without “ping-pong”
  - Also, long lines without false sharing



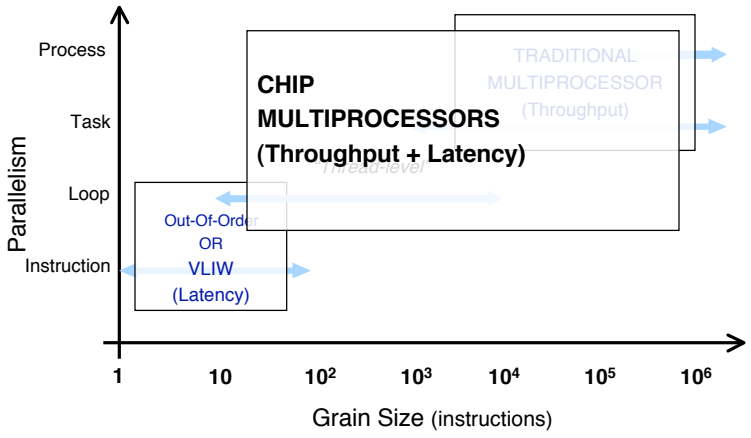
Shared Cache Disadvantages

- Fundamental BW limitation
- Increases latency of all accesses
  - Must go through crossbar
  - Larger cache
  - L1 hit time determines proc. cycle time !!!
- Potential for negative interference
  - One proc flushes data needed by another
  - Bad for completely independent tasks

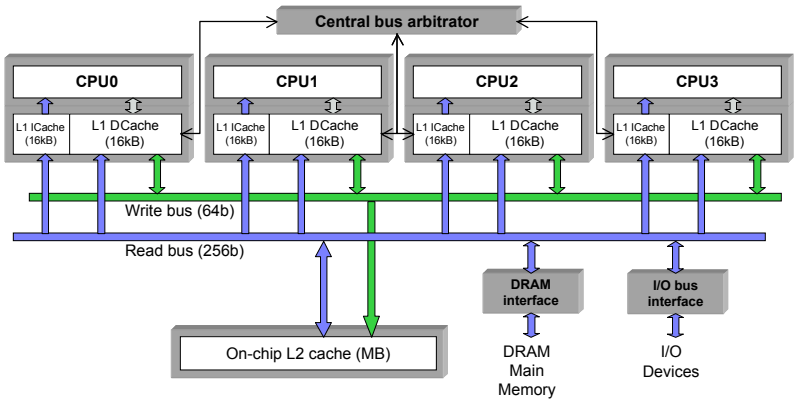


- Many L2 caches are shared today
  - IBM Power5, Sun Niagara
  - Allows sharing, but doesn't affect L1 hit time

Parallelism, Latency and Throughput

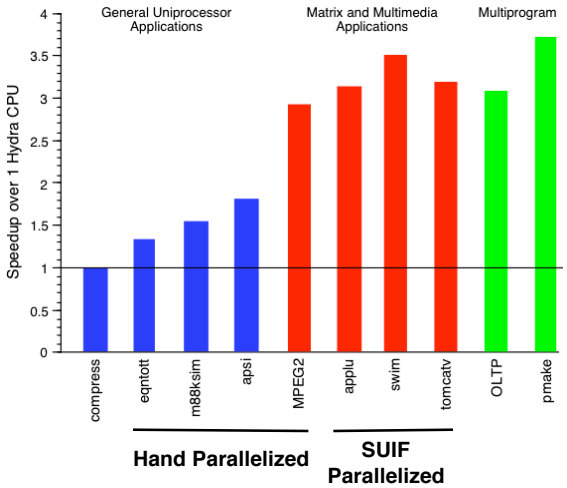


Stanford Hydra Chip-Multiprocessor



- **Large shared L2 cache**
- **Low-latency interprocessor communication (10-15 cycles)**

Parallel Performance



- Varying levels of system performance
  - Throughput workloads work well
  - Very parallel apps (matrix-based FP and multimedia) are excellent
  - Acceptable only with *some* less parallel (integer) apps

Need to Parallelize Applications

- Parallel software for single applications is limited
  - Hand-parallelized applications
  - Auto-parallelized dense matrix FORTRAN applications
- Traditional auto-parallelization of general purpose-programs is very difficult
  - Synchronization required for correctness
  - General programs complicate dependence analysis
    - Random pointers in C code
  - Compile time analysis is too conservative
- How can hardware help?
  - Low latency = small threads w/ lots of communication OK
  - Reduce need for pointer disambiguation
  - Allow the compiler to be aggressive

Solution: Thread-level Speculation

- TLS enables parallelization without regard for data-dependencies
  - Loads and stores follow original sequential semantics
  - Speculation hardware ensures correctness
  - Add synchronization only for performance
- Parallelization is now easy
  - Loop parallelization can be automated
  - Break code into arbitrary threads
- Data speculation support
  - Wisconsin Multiscalar, CMU Stampede, Illinois Torrelas group
  - Hydra CMP provides low-overhead support for TLS+Dynamic compiler+new programming model

Midterm