Computer Science 146
Computer Architecture

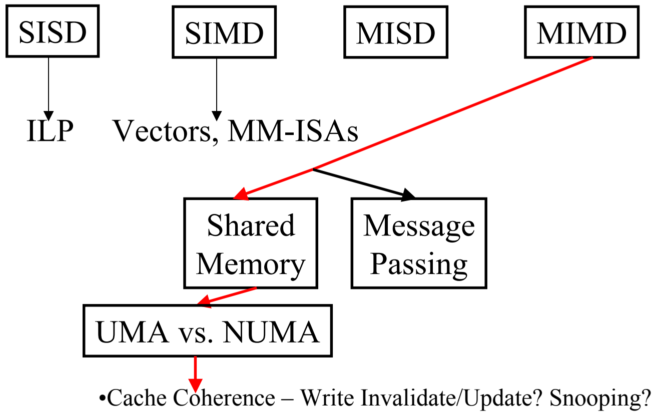
Fall 2019

Harvard University

Instructor: Prof. David Brooks
dbrooks@eecs.harvard.edu

Lecture 20: More Multiprocessors

Computation Taxonomy



- Cache Coherence – Write Invalidate/Update? Snooping? Directory?
- Synchronization
- Memory Consistency

Major issues for Shared Memory

- Cache coherence
  - "Common Sense"
    - P1 Read[X] => P1 Write[X] => P1 Read[X] will return X
    - P1 Write[X] => P2 Read[X] => will return value written by P1
    - P1 Write[X] => P2 Write[X] => Serialized (all processor see the writes in the same order)
- Synchronization
  - Atomic read/write operations
- Memory consistency Model
  - *When* will a written value be seen?
  - P1 Write[X] (10ps later) P2 Read[X] what happens?
- These are not issues for message passing systems
  - Why?

Cache Coherence

- Benefits of coherent caches in parallel systems?
  - Migration and Replication of shared data
  - Migration: data moved locally lowers latency + main memory bandwidth
  - Replication: data being simultaneously read can be replicated to reduce latency + contention
- Problem: sharing of writeable data

Processor 0	**Processor 1**	**Correct value of A in:**
		Memory
Read A		Memory, p0 cache
	Read a	Memory, p0 cache, p1 cache
Write A		P0 cache, memory (if write-through)
	Read A	P1 gets stale value on hit

Solutions to Cache Coherence

- No caches
  - Not likely :)
- Make shared data non-cacheable
  - Simple software solution
  - low performance if lots of shared data
- Software flush at strategic times
  - Relatively simple, but could be costly (frequent syncs)
- Hardware cache coherence
  - Make memory and caches coherent/consistent with each other

HW Coherence Protocols

- Absolute coherence
  - All copies of each block have same data at all times
  - A little bit overkill…
- Need *appearance* of absolute coherence
  - Temporary incoherence is ok
    - Similar to write-back cache
    - As long as all loads get their correct value
- Coherence protocol: FSM that runs at every cache
  - Invalidate protocols: invalidate copies in other caches
  - Update protocols: update copies in other caches
  - Snooping vs. Directory based protocols (HW implementation)
  - Memory is always updated

Write Invalidate

- Much more common for most systems
- Mechanics
  - Broadcast address of cache line to invalidate
  - All processor snoop until they see it, then invalidate if in local cache
  - Same policy can be used to service cache misses in write-back caches

Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Invalidation miss for X	1		0
CPU B reads X	Cache miss for X	1	1	1

Write Update (Broadcast)

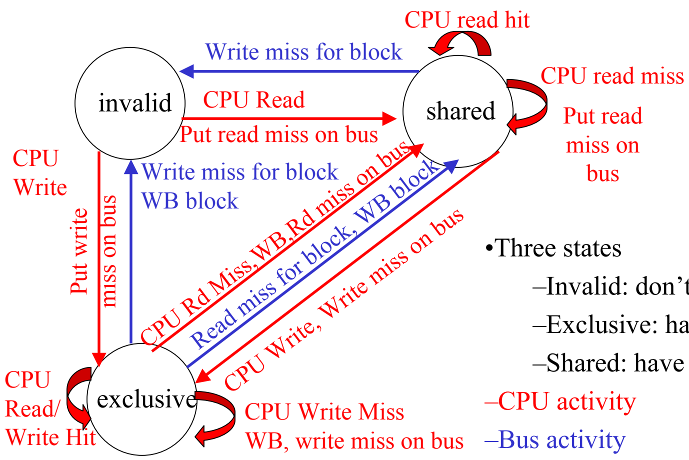
Processor Activity	Bus Activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of Memory Location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes 1 to X	Write Broadcast for X	1	1	1
CPU B reads X		1	1	1

- Bandwidth requirements are excessive
  - Can reduce by checking if word is shared (also can reduce write-invalidate traffic)
- Comparison with Write invalidate
  - Multiple writes, no interveaning reads require multiple broadcasts
  - Multiple broadcasts needed for multiple word cache line writes (only 1 invalidate)
  - Advantage: other processors can get the data faster

Bus-based protocols (Snooping)

- Snooping
  - All caches see and react to *all* bus events
  - Protocol relies on global visibility of events (ordered broadcast)
- Events:
  - Processor (events from own processor)
    - Read (R), Write (W), Writeback (WB)
  - Bus Events (events from other processors)
    - Bus Read (BR), Bus Write (BW)

Three-State Invalidate Protocol

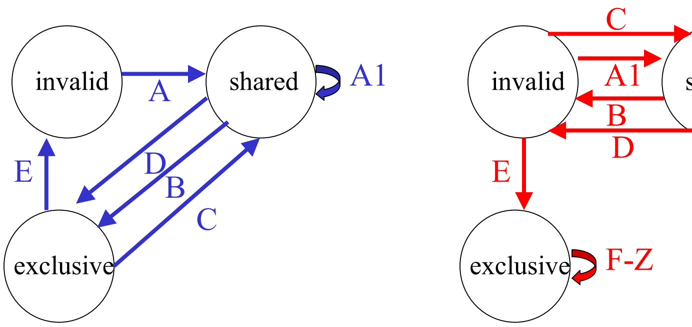


- Three states
  - Invalid: don't have block
  - Exclusive: have block and wrote it
  - Shared: have block but only read it
  - CPU activity
  - Bus activity

Three-State Invalidate Protocol

Processor 1

Processor 2



A:	P1 Read
A1:	P2 Read
B:	P1 Write
C:	P2 Read
D:	P1 Write
E:	P2 Write
F-Z:	P2 Write

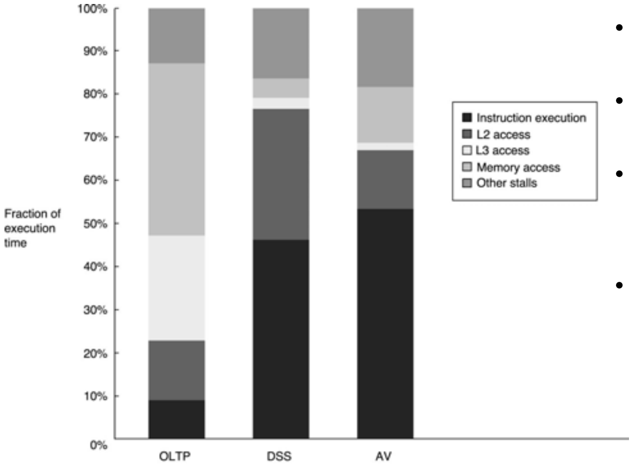
Three State Invalidate Problems

- Real implementations are much more complicated
  - Assumed all operations are *atomic*
    - Multiphase operation can be done with no intervening ops
    - E.g. write miss detected, acquire bus, receive response
    - Even read misses are non-atomic with split transaction busses
    - Deadlock is a possibility, see Appendix I
  - Other Simplifications
    - Write hits/misses treated same
    - Don’t distinguish between true sharing and clean in one cache
    - Other caches could supply data on misses to shared block

Scalable Coherence Protocols: Directory Based Systems

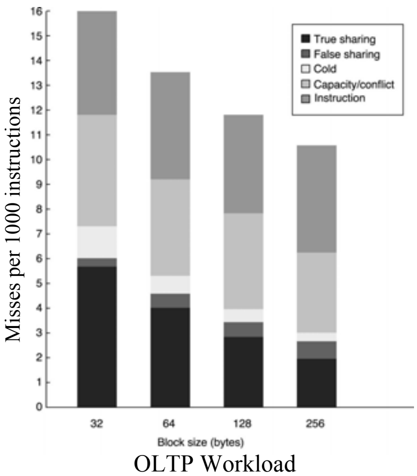
- Bus based systems are not scalable
  - Not enough bus B/W for everyone’s coherence traffic
  - Not enough processor snooping B/W to handle everyone’s traffic
- Directories: scalable cache coherence for large MPs
  - Each memory entry (cache line) has a bit vector (1 bit per processor)
  - Bit vector tracks which processors have cached copies of line
  - Send all messages to memory directory
  - If no other cached copies, memory returns data
  - Otherwise memory forwards request to correct processor
    - + Low b/w consumption (communicate only with relevant processors)
    - + Works with general interconnect (bus not needed)
    - - Longer latency (3-hop transaction: p0 => directory => p1 => p0)

Multiprocessor Performance



- OLTP: TPC-C (Oracle)
  - CPI ~ 7.0
- DSS: TPC-D
  - CPI ~ 1.61
- AV: Altavista
  - CPI ~1.3
- Huge fraction of time in memory and I/O for TPC-C

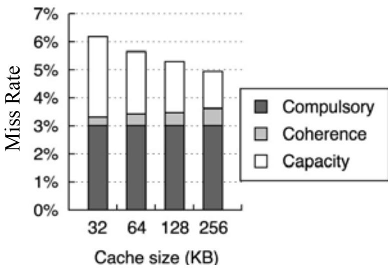
Coherence Protocols: Performance



OLTP Workload

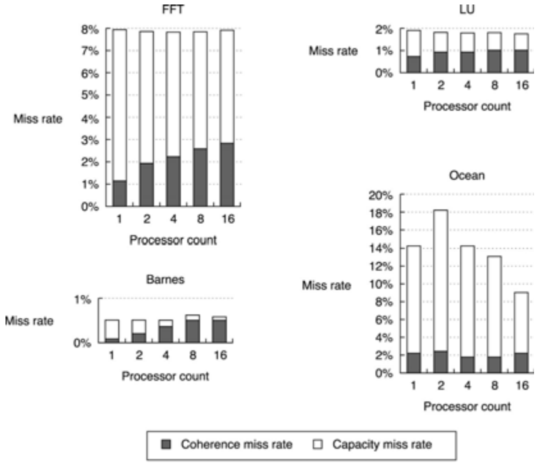
- As block size is increased
  - Coherence misses increase (false sharing)
  - False sharing
    - Sharing of different data in the same cache line
    - *Block* is shared, but *data word* is not shared

Coherence Protocols: Performance



- 3C Miss model => 4C miss model
  - Capacity, compulsory, conflict
  - *Coherence*: additional misses due to coherence protocol
  - Complicated uniprocessor cache analysis
- As cache size is increased
  - Capacity misses decrease
  - Coherence misses may increase (more shared data is cached)

Coherence Protocols: Performance



- As processors are added
  - Coherence misses increase (more communication)

Synchronization

- *Synchronization*: important issue for shared memory
  - Regulates access to shared data
  - E.g. semaphore, monitor, critical section (s/w constructs)
  - Synchronization primitive: *lock*
  - Well written parallel programs will synchronize on shared data

```
acquire(lock);       //while (lock != 0); lock = 1;
critical section;
release(lock);       // lock = 0;

0: ldw r1, lock      // wait for lock to be free
1: bnez r1, #0
2: stw #1, lock      // acquire lock
…                    // critical section
9: stw #0, lock      // release lock
```

Implementing Locks

- Called spin lock, doesn't work quite right…

```
  Processor 0           Processor 1
0: ldw r1, lock
1: bnez r1, #0                                // p0 sees lock free
                      0: ldw r1, lock
                      1: bnez r1, #0          // p1 sees lock free
2: stw #1, lock                               // p0 acquires lock
                      2: stw #1, lock         // p1 acquires lock
…                                             // p0 and p1 in
                      …                       // critical section
…                                             // together
9: stw #0, lock
```

Implementing Locks

- Problem: acquire sequence (load-test-store) is not atomic
  - Solution: ISA provides an atomic lock-acquire operation
  - Load+Check+Store in one instruction (uninterruptible by definition)
  - E.g. test+set instruction (fetch-and-increment, exchange)

```
t&s r1, lock   // ldw r1, lock; stw #1, lock
```

```
0: t&s r1, lock
1: bnez r1, 0
2: …
3: stw r1, #0
```

Lock-release is already atomic

Memory Ordering Consistency

- Memory updates may become reordered by the memory system

```
Processor 0            Processor 1
A = 0;                 B = 0;
A = 1;                 B = 1;
L1: if (B==0)          L2: if (A==0)
Critical section       Critical Section
```

- Intuitively, impossible for both to be in critical section
- BUT, if memory ops are reordered it could happen
- Coherence: A’s and B’s must be same eventually
- Doesn’t specify relative timing of coherence

Memory Ordering: Sequential Consistency

- “System is sequentially consistent if the result of ANY execution is the same as if the operations of all processors were executed in SOME sequential order and the operations of each individual processor appear in this sequence in program order” [Lamport]
- Sequential Consistency
  - All loads and stores in order
  - Delay completion of memory access until all invalidations caused by that access complete
  - Delay next memory access until previous one completes
  - Delay read of A/B until previous write of A/B completes
    - Cannot place writes in a write buffer and continue with read!
  - Simple for programmer
  - Not much room for HW/SW performance optimizations

Sequential Consistency Performance

- Suppose a write miss takes:
  - 40 cycles to establish ownership
  - 10 cycles to issue each invalidate after ownership
  - 50 cycles to complete/acknowledge invalidates
- Suppose 4 processors have shared cache block

- Sequential Consistency: Must wait 40 + 10 + 10 + 10 + 10 + 50 cycles = 130 cycles
- Ownership time is only 40 cycles
- Write buffers could continue before this

Weaker Consistency Models

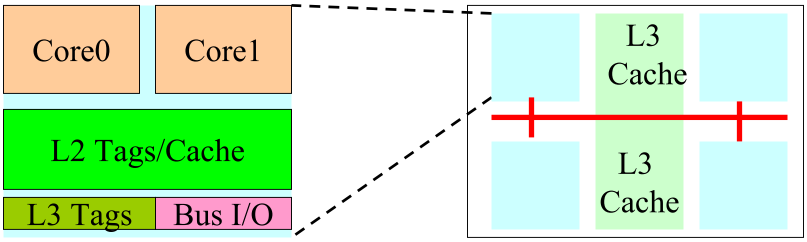
- Assume programs are synchronized
- Observation: SC only needed for *lock* variables
  - Other variables?
  - Either in critical section (no parallel access)
  - Or not shared
- Weaker consistency: can delay/reorder loads and stores
  - More room for hardware optimizations
  - Somewhat trickier programming model

One other possibility…

- Many processors have extensive support for speculation recovery
- Can this be re-used to support strict consistency?
  - Use delayed commit feature for memory references
  - If we receive an invalidate message, back-out the computation and restart
  - Rarely triggered anyway (only on unsynchronized accesses that cause a race)

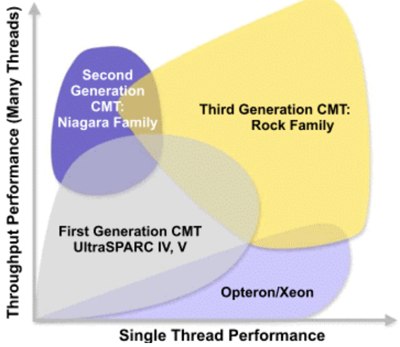
Today multiprocessor trends: Single Chip multiprocessors

- e.g. IBM POWER4
  - 1 chip contains: 2 1.3GHz processors, L2, L3 tags
  - Can connect 4 chips on 1 MCM to create 8-way system
  - Targets threaded server workloads, high-performance computing



Single Chip MP: The Future – but how many cores?

- Some companies are betting big on it…
  - Sun's Niagara (8 cores per chip)
  - Intel's rumored Tukwila or Tanglewood processor (8 or 16?)
- Basis in Piranha research project (ISCA 2000)


Throughput Performance (Many Threads)
Second Generation CMT: Niagara Family
Third Generation CMT: Rock Family
First Generation CMT UltraSPARC IV, V
Opteron/Xeon
Single Thread Performance

Multithreading

- Another trend: multithreaded processors
  - Processor utilization: IPC / Processor Width
    - Decreases as width increases (~50% on 4-wide)
    - Why? Cache misses, branch mis-predicts, RAW dependencies
  - Idea? Two (or more) processes (threads) share one pipeline
  - Replicate process (thread) state
    - PC, register file, bpred, page table pointer, etc (5% area overhead)
  - One copy of stateless (naturally tagged) structures
    - Caches, functional units, buses, etc
  - Hardware thread context must be fast
    - Multiple on-chip contexts => no need to load from memory

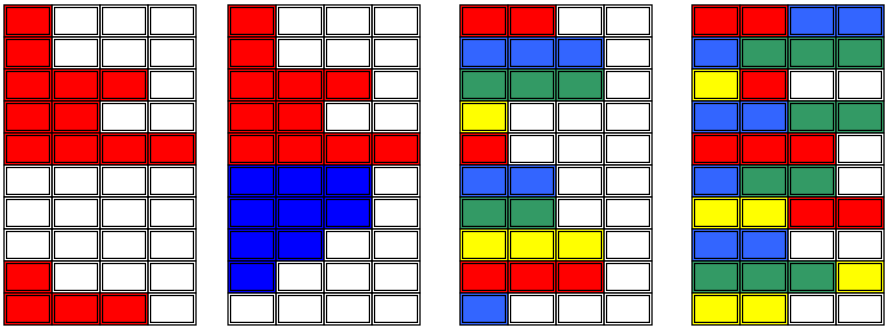
Multithreading Paradigms

SuperScalar

Coarse MT

Fine MT

Simultaneous MT



Pentium 3,
Alpha EV6/7

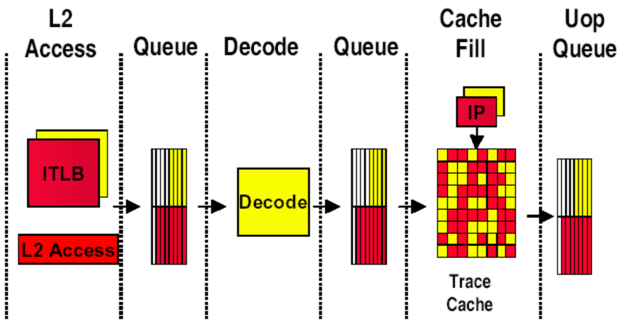
IBM Pulsar

Intel P4-HT,
EV8,POWER5

Coarse vs. Fine Grained MT

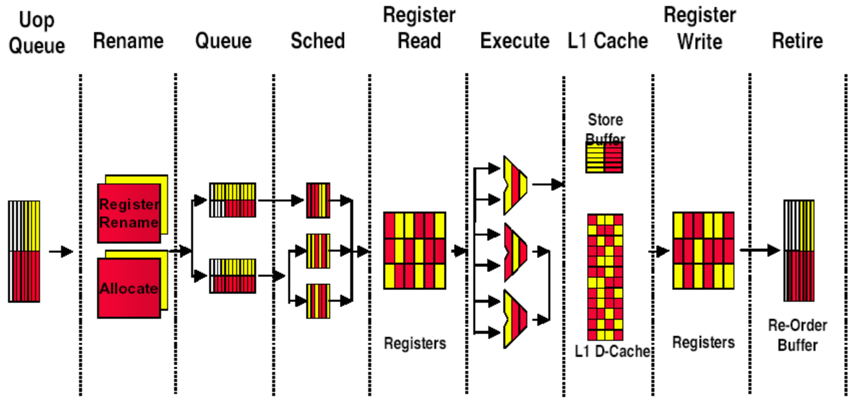
- Coarse-Grained
  - Makes sense for in-order/shorter pipelines
  - Switch threads on long stalls (L2 cache misses)
  - Threads don’t interfere with each other much
  - Can’t improve utilization on L1 misses/bpred mispredicts
- Fine-grained
  - Out-of-order, deep pipelines
  - Instructions from multiple threads in stage at a time, miss or not
  - Improves utilization in all scenarios
  - Individual thread performance suffers due to interference

Pentium 4 Front-End



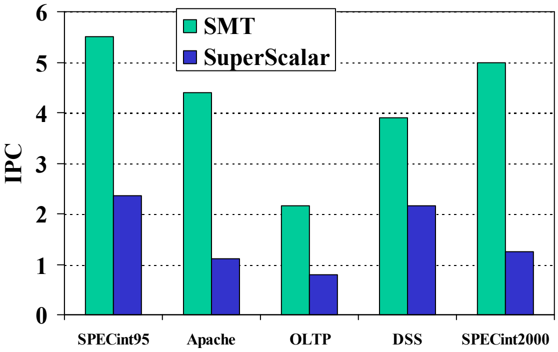
- Front end resources arbitrate between threads every cycle
- ITLB, RAS, BHB(Global History), Decode Queue are duplicated

Pentium 4 Backend



- Some queues and buffers are partitioned (only ½ entries per thread)
- Scheduler is oblivious to instruction thread Ids (limit on # per scheduler)

Multithreading Performance



9-stage pipeline
128KB I/D Cache
6 IntALU (4 Load/Store)
4 FPALUs
8 SMT threads
8-insn fetch (2 threads)

- Intel claiming more like 20-30% increase on P4
- Lot of debate exists on performance benefits

Next Lectures

- Disks and I/O
- Clusters
- Novel and Non-CPU processors