CIS 501
Computer Architecture

Unit 9: Multicore
(Shared Memory Multiprocessors)

This Unit: Shared Memory Multiprocessors

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models

Readings
- H+P
  - Chapter 4

Multiplying Performance
- A single processor can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism
  - Limited cache hierarchy

- What if we need even more computing power?
  - Use multiple processors!
  - But how?

- High-end example: Sun Ultra Enterprise 25k
  - 72 UltraSPARC IV+ processors, 1.5Ghz
  - 1024 GBs of memory
  - Niche: large database servers
  - $$$
Multicore: Mainstream Multiprocessors

- **Multicore chips**
- **IBM Power5**
  - Two 2+GHz PowerPC cores
  - Shared 1.5 MB L2, L3 tags
- **AMD Quad Phenom**
  - Four 2+ GHz cores
  - Per-core 512KB L2 cache
  - Shared 2MB L3 cache
- **Intel Core 2 Quad**
  - Four cores, shared 4 MB L2
  - Two 4MB L2 caches
- **Sun Niagara**
  - 8 cores, each 4-way threaded
  - Shared 2MB L2, shared FP
  - For servers, not desktop

Why multicore? What else would you do with 500 million transistors?

Application Domains for Multiprocessors

- **Scientific computing/supercomputing**
  - Examples: weather simulation, aerodynamics, protein folding
  - Large grids, integrating changes over time
  - Each processor computes for a part of the grid
- **Server workloads**
  - Example: airline reservation database
  - Many concurrent updates, searches, lookups, queries
  - Processors handle different requests
- **Media workloads**
  - Processors compress/decompress different parts of image/frames
- **Desktop workloads...**
- **Gaming workloads...**
  - But software must be written to expose parallelism

But First, Uniprocessor Concurrency

- **Software “thread”**
  - Independent flow of execution
  - Context state: PC, registers
  - Threads generally share the same memory space
  - "Process" like a thread, but different memory space
  - Java has thread support built in, C/C++ supports P-threads library

- **Generally, system software (the O.S.) manages threads**
  - "Thread scheduling", "context switching"
  - All threads share the one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in CIS380

Multithreaded Programming Model

- **Programmer explicitly creates multiple threads**
- **All loads & stores to a single shared memory space**
  - Each thread has a private stack frame for local variables
- **A “thread switch” can occur at any time**
  - Pre-emptive multithreading by OS
- **Common uses:**
  - Handling user interaction (GUI programming)
  - Handling I/O latency (send network message, wait for response)
  - Expressing parallel work via Thread-Level Parallelism (TLP)
Aside: Hardware Multithreading

- **Hardware Multithreading (MT)**
  - Multiple threads dynamically share a single pipeline (caches)
  - Replicate thread contexts: PC and register file
  - **Coarse-grain MT**: switch on L2 misses  **Why?**
  - **Simultaneous MT**: no explicit switching, fine-grain interleaving
    - Pentium4 is 2-way hyper-threaded, leverages out-of-order core
    - MT Improves utilization and throughput
    - Single programs utilize <50% of pipeline (branch, cache miss)
  - MT does not improve single-thread performance
  - Individual threads run as fast or even slower

Simplest Multiprocessor

- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share caches (we'll address this bottleneck later)
- Same “shared memory” or “multithreaded” model
  - Loads and stores from two processors are interleaved
  - Advantages/disadvantages over hardware multithreading?

Shared Memory Implementations

- **Multiplexed uniprocessor**
  - Runtime system and/or OS occasionally pre-empt & swap threads
  - Interleaved, but no parallelism

- **Hardware multithreading**
  - Tolerate pipeline latencies, higher efficiency
  - Same interleaved shared-memory model

- **Multiprocessing**
  - Multiply execution resources, higher peak performance
  - Same interleaved shared-memory model
  - Foreshadowing: allow private caches, further disentangle cores

- **All have same shared memory programming model**

Shared Memory Issues

- Three in particular, not unrelated to each other

- **Synchronization**
  - How to regulate access to shared data?
  - How to implement critical sections?

- **Cache coherence**
  - How to make writes to one cache “show up” in others?

- **Memory consistency model**
  - How to keep programmer sane while letting hardware optimize?
  - How to reconcile shared memory with out-of-order execution?
Example: Parallelizing Matrix Multiply

```c
for (I = 0; I < 100; I++)
for (J = 0; J < 100; J++)
for (K = 0; K < 100; K++)
```

- How to parallelize matrix multiply over 100 processors?
- One possibility: give each processor 100 iterations of I

```c
for (J = 0; J < 100; J++)
for (K = 0; K < 100; K++)
```
- Each processor runs copy of loop above
- `my_id()` function gives each processor ID from 0 to N
- Parallel processing library provides this function

Example: Thread-Level Parallelism

```
struct acct_t { int bal; }
shared struct acct_t accts[MAX_ACCT];
int id, amt;
if (accts[id].bal >= amt)
    { accts[id].bal -= amt;
     give_cash();
    }
```

- **Thread-level parallelism (TLP)**
  - Collection of asynchronous tasks: not started and stopped together
  - Data shared "loosely" (sometimes yes, mostly no), dynamically
  - Example: database/web server (each query is a thread)
  - `accts` is shared, can't register allocate even if it were scalar
  - `id` and `amt` are private variables, register allocated to `r1, r2`
- Running example

### Example: An Example Execution

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Mem</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td>500</td>
<td>0</td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: b1t r4,r2,6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>5: call give_cash</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Mem</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: b1t r4,r2,6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>5: call give_cash</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Two $100 withdrawals from account #241 at two ATMs
  - Each transaction maps to thread on different processor
  - Track `accts[241].bal` (address is in `r3`)

### Example: An A Problem Execution

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Mem</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>500</td>
<td></td>
</tr>
<tr>
<td>2: b1t r4,r2,6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: call give_cash</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Mem</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: b1t r4,r2,6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>5: call give_cash</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Problem: wrong account balance! Why?
  - Solution: synchronize access to account balance

CIS 501 (Martin/Roth): Multicore
Synchronization

- **Synchronization**: a key issue for shared memory
  - Regulate access to shared data (mutual exclusion)
  - Software constructs: semaphore, monitor, mutex
  - Low-level primitive: **lock**
    - Operations: acquire(lock) and release(lock)
    - Region between acquire and release is a **critical section**
    - Must interleave acquire and release
    - Interfering acquire will block

```c
struct acct_t { int bal; }
shared struct acct_t accts[MAX_ACCT];
shared int lock;
int id, amt;
acquire(lock);
if (accts[id].bal >= amt) {
  accts[id].bal -= amt;
give_cash();
} release(lock);
```

A Synchronized Execution

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>call acquire(lock)</td>
<td>call acquire(lock)</td>
</tr>
<tr>
<td>0: addi r1,accts,r3</td>
<td>0: addi r1,accts,r3</td>
</tr>
<tr>
<td>1: blt r4,r2,6</td>
<td>1: blt r4,r2,6</td>
</tr>
<tr>
<td>2: sub r4,r2,r4</td>
<td>3: sub r4,r2,r4</td>
</tr>
<tr>
<td>&lt;&lt;&lt; Interrupt &gt;&gt;</td>
<td>&lt;&lt;&lt; Interrupt &gt;&gt;</td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>4: st r4,0(r3)</td>
</tr>
<tr>
<td>call release(lock)</td>
<td>call release(lock)</td>
</tr>
<tr>
<td>5: call give_cash</td>
<td>5: call give_cash</td>
</tr>
<tr>
<td>Mem</td>
<td>Time</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
</tr>
</tbody>
</table>

Strawman Lock (Incorrect)

- **Spin lock**: software lock implementation
  - acquire(lock): while (lock != 0): lock = 1;
    - “Spin” while lock is 1, wait for it to turn 0
  - release(lock): lock = 0;
    - R0: st r0,0(&lock) // r0 holds 0

Strawman Lock (Incorrect)

- Spin lock makes intuitive sense, but doesn’t actually work
  - Loads/stores of two acquire sequences can be interleaved
  - Lock acquire sequence also not atomic
  - Same problem as before!

- Note, release is trivially atomic
A Correct Implementation: SYSCALL Lock

**ACQUIRE_LOCK:**
1. disable_interrupts
2. ld r6,0(&lock)
3. bnez r6,#A0
4. addi r6,1,r6
5. st r6,0(&lock)
6. enable_interrupts
7. return

• Implement lock in a SYSCALL
  • Only kernel can control interleaving by disabling interrupts
  + Works...
  – Large system call overhead
  – But not in a hardware multithreading or a multiprocessor...

Better Spin Lock: Use Atomic Swap

• ISA provides an atomic lock acquisition instruction
  • Example: atomic swap
    
    ```
    mov r1->r2
    ld r1,0(&lock)
    st r2,0(&lock)
    ```

  • New acquire sequence
    (value of r1 is 1)
    1. swap r1,0(&lock)
    2. bnez r1,A0
    3. If lock was initially busy (1), doesn’t change it, keep looping
    4. If lock was initially free (0), acquires it (sets it to 1), break loop

  • Insures lock held by at most one thread
  • Other variants: exchange, compare-and-swap, test-and-set (t&s), or fetch-and-add

Atomic Update/Swap Implementation

• How is atomic swap implemented?
  • Need to ensure no intervening memory operations
  • Requires blocking access by other threads temporarily (yuck)

• How to pipeline it?
  • Both a load and a store (yuck)
  • Not very RISC-like
  • Some ISAs provide a “load-link” and “store-conditional” insn. pair

RISC Test-And-Set

• t&s: a load and store in one insn is not very “RISC”
  • Broken up into micro-ops, but then how are mops made atomic?

• ll/sc: load-locked / store-conditional
  • Atomic load/store pair
    ```
    ll r1,0(&lock)
    // potentially other insn
    sc r2,0(&lock)
    ```
  • On ll, processor remembers address...
    • ...And looks for writes by other processors
  • If write is detected, next sc to same address is annulled
    • Sets failure condition
Lock Correctness

+ Test-and-set lock actually works...
  • Thread 1 keeps spinning

Test-and-Set Lock Performance

- ...but performs poorly
  • Consider 3 processors rather than 2
  • Processor 2 (not shown) has the lock and is in the critical section
  • But what are processors 0 and 1 doing in the meantime?
    • Loops of \texttt{t&s}, each of which includes a \texttt{st}
      - Repeated stores by multiple processors costly (more in a bit)
      - Generating a ton of useless interconnect traffic

Test-and-Test-and-Set Locks

• Solution: \texttt{test-and-test-and-set locks}
  • New acquire sequence
    \begin{align*}
    \text{A0: } & \text{ld r1,0(&lock)} \\
    \text{A1: } & \text{bnez r1,#A0} \\
    \text{A2: } & \text{addi r1,1,r1} \\
    \text{A3: } & \text{t&s r1,0(&lock)} \\
    \text{A4: } & \text{bnez r1,#A0}
    \end{align*}
  • Within each loop iteration, before doing a \texttt{t&s}
    - Spin doing a simple test (\texttt{ld}) to see if lock value has changed
    - Only do a \texttt{t&s} (\texttt{st}) if lock is actually free
  • Processors can spin on a busy lock locally (in their own cache)
    + Less unnecessary interconnect traffic
  • Note: test-and-test-and-set is not a new instruction!
    • Just different software

Queue Locks

• Test-and-test-and-set locks can still perform poorly
  • If lock is contended for by many processors
  • Lock release by one processor, creates “free-for-all” by others
    - Interconnect gets swamped with \texttt{t&s} requests
  • \textbf{Software queue lock}
    • Each waiting processor spins on a different location (a queue)
    • When lock is released by one processor...
      - Only the next processors sees its location go “unlocked”
      - Others continue spinning locally, unaware lock was released
    • Effectively, passes lock from one processor to the next, in order
      + Greatly reduced network traffic (no mad rush for the lock)
      + Fairness (lock acquired in FIFO order)
      - Higher overhead in case of no contention (more instructions)
      - Poor performance if one thread gets swapped out
Programming With Locks Is Tricky

- Multicore processors are the way of the foreseeable future
  - thread-level parallelism anointed as parallelism model of choice
  - Just one problem...

- Writing lock-based multi-threaded programs is tricky!

  - More precisely:
    - Writing programs that are correct is "easy" (not really)
    - Writing programs that are highly parallel is "easy" (not really)
      - Writing programs that are both correct and parallel is difficult
        - And that's the whole point, unfortunately
    - Selecting the "right" kind of lock for performance
      - Spin lock, queue lock, ticket lock, read/writer lock, etc.

  - Locking granularity issues

Coarse-Grain Locks: Correct but Slow

- **Coarse-grain locks**: e.g., one lock for entire database
  + Easy to make correct: no chance for unintended interference
  - Limits parallelism: no two critical sections can proceed in parallel

```c
struct acct_t { int bal; };  
shared struct acct_t  accts[MAX_ACCT];  
int id,amt;  
shared int lock;  

acquire(lock);  
if (accts[id].bal >= amt) {  
    accts[id].bal -= amt;  
give_cash(); }  
release(lock);  
```

Fine-Grain Locks: Parallel But Difficult

- **Fine-grain locks**: e.g., multiple locks, one per record
  + Fast: critical sections (to different records) can proceed in parallel
  - Difficult to make correct: easy to make mistakes
    - This particular example is easy
      - Requires only one lock per critical section
    - Consider critical section that requires two locks...

```c
struct acct_t { int bal,lock; };  
shared struct acct_t  accts[MAX_ACCT];  
int id,amt;  

acquire(accts[id].lock);  
if (accts[id].bal >= amt) {  
    accts[id].bal -= amt;  
give_cash(); }  
release(accts[id].lock);  
```

Multiple Locks

- **Multiple locks**: e.g., acct-to-acct transfer
  - Must acquire both id_from, id_to locks
  - Running example with accts 241 and 37
  - Simultaneous transfers 241 → 37 and 37 → 241
  - Contrived... but even contrived examples must work correctly too

```c
struct acct_t { int bal,lock; };  
shared struct acct_t  accts[MAX_ACCT];  
int id_from,id_to,amt;  

acquire(accts[id_from].lock);  
acquire(accts[id_to].lock);  
if (accts[id_from].bal >= amt) {  
    accts[id_from].bal -= amt;  
    accts[id_to].bal += amt; }  
release(accts[id_to].lock);  
release(accts[id_from].lock);  
```
**Multiple Locks And Deadlock**

- **Deadlock**: circular wait for shared resources
  - Thread 0 has lock 241 waits for lock 37
  - Thread 1 has lock 37 waits for lock 241
  - Obviously this is a problem
  - The solution is ...

```c
Thread 0
id_from = 241;
id_to = 37;
acquire(accts[241].lock);
// wait to acquire lock
37
// waiting...
// still waiting...

Thread 1
id_from = 37;
id_to = 241;
acquire(accts[37].lock);
// wait to acquire lock 241
37
// waiting...
// still waiting...
```

**Correct Multiple Lock Program**

- **Always acquire multiple locks in same order**
  - Just another thing to keep in mind when programming

```c
struct acct_t { int bal, lock; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;
int id_first = min(id_from, id_to);
int id_second = max(id_from, id_to);
acquire(accts[id_first].lock);
acquire(accts[id_second].lock);
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
}
release(accts[id_second].lock);
release(accts[id_first].lock);
```

**Correct Multiple Lock Execution**

```c
Thread 0
id_from = 241;
id_to = 37;
id_first = min(241,37)=37;
id_second = max(37,241)=241;
acquire(accts[37].lock);
acquire(accts[241].lock);
// do stuff
// release(accts[241].lock);
release(accts[37].lock);
acquire(accts[37].lock);
```

**More Lock Madness**

- What if...
  - Some actions (e.g., deposits, transfers) require 1 or 2 locks...
  - ...and others (e.g., prepare statements) require all of them?
  - Can these proceed in parallel?
- What if...
  - There are locks for global variables (e.g., operation id counter)?
  - When should operations grab this lock?
- What if... what if... what if...

**So lock-based programming is difficult...**

**...wait, it gets worse**
And To Make It Worse...

- **Acquiring locks is expensive...**
  - By definition requires a slow atomic instructions
  - Specifically, acquiring write permissions to the lock
  - Ordering constraints (see soon) make it even slower

- **...and 99% of the time unnecessary**
  - Most concurrent actions don’t actually share data
    - You paying to acquire the lock(s) for no reason

- Fixing these problem is an area of active research
  - One proposed solution “Transactional Memory”

Research: Transactional Memory (TM)

- **Transactional Memory**
  - Programming simplicity of coarse-grain locks
  - Higher concurrency (parallelism) of fine-grain locks
    - Critical sections only serialized if data is actually shared
  - No lock acquisition overhead
  - Hottest thing since sliced bread (or was a few years ago)
  - No fewer than 9 research projects: Brown, Stanford, MIT, Intel...
    - Penn too

Transactional Memory: The Big Idea

- **Big idea I: no locks, just shared data**
  - Look ma, no locks

- **Big idea II: optimistic (speculative) concurrency**
  - Execute critical section speculatively, abort on conflicts
  - “Better to beg for forgiveness than to ask for permission”

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
} end_transaction();
```

Transactional Memory: Read/Write Sets

- **Read set:** set of shared addresses critical section reads
  - Example: accts[37].bal, accts[241].bal

- **Write set:** set of shared addresses critical section writes
  - Example: accts[37].bal, accts[241].bal

```c
struct acct_t { int bal; };
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
} end_transaction();
```
Transactional Memory: Begin

- `begin_transaction`
  - Take a local register checkpoint
  - Begin locally tracking read set (remember addresses you read)
    - See if anyone else is trying to write it
  - Locally buffer all of your writes (invisible to other processors)
  + Local actions only: no lock acquire

```c
struct acct_t { int bal; }
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
}
end_transaction();
```

Transaction Memory Implementation

- How are read-set/write-set implemented?
  - Track locations accessed using bits in the cache

- Read-set: additional “transactional read” bit per block
  - Set on reads between begin_transaction and end_transaction
  - Any other write to block with set bit \( \Rightarrow \) triggers abort
  - Flash cleared on transaction abort or commit

- Write-set: additional “transactional write” bit per block
  - Set on writes between begin_transaction and end_transaction
  - Before first write, if dirty, initiate writeback ("clean" the block)
  - Flash cleared on transaction commit
  - On transaction abort: blocks with set bit are invalidated

Transactional Memory: End

- `end_transaction`
  - Check read set: is all data you read still valid (i.e., no writes to any)
  - Yes? Commit transactions: commit writes
  - No? Abort transaction: restore checkpoint

```c
struct acct_t { int bal; }
shared struct acct_t accts[MAX_ACCT];
int id_from, id_to, amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
}
end_transaction();
```

Transactional Execution

**Thread 0**

```c
id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
    ...
    // write accts[241].bal
    // abort
}
end_transaction();
```

**Thread 1**

```c
id_from = 37;
id_to = 241;

begin_transaction();
if(accts[37].bal > 100) {
    accts[37].bal -= amt;
    accts[241].bal += amt;
}
end_transaction();
// no writes to accts[241].bal
// no writes to accts[37].bal
// commit```
Transactional Execution II (More Likely)

Thread 0

id_from = 241;
id_to = 37;

begin_transaction();
if(accts[241].bal > 100) {
  accts[241].bal -= atm;
  acts[37].bal += atm;
}
end_transaction();

// no write to accts[240].bal
// no write to accts[37].bal
// commit

Thread 1

id_from = 450;
id_to = 118;

begin_transaction();
if(accts[450].bal > 100) {
  accts[450].bal -= atm;
  acts[118].bal += atm;
}
end_transaction();

// no write to accts[450].bal
// no write to accts[118].bal
// commit

• Critical sections execute in parallel

So, Let’s Just Do Transactions?

• What if...
  - Read-set or write-set bigger than cache?
  - Transaction gets swapped out in the middle?
  - Transaction wants to do I/O or SYSCALL (not-abortable)?

• How do we transactify existing lock based programs?
  - Replace acquire with begin_trans does not always work

• Several different kinds of transaction semantics
  - Are transactions atomic relative to code outside of transactions?

• Do we want transactions in hardware or in software?
  - What we just saw is hardware transactional memory (HTM)

• That’s what these research groups are looking at
  - Sun’s Rock processor has best-effort hardware TM
  - Speculative locking: Azul systems and Intel (rumor)

In The Meantime: Do SLE

Processor 0

acquire(accts[37].lock); // don’t actually set lock to 1
// begin tracking read/write sets
// CRITICAL_SECTION
// check read set
// no conflicts? Commit, don’t actually set lock to 0
// conflicts? Abort, retry by acquiring lock
release(accts[37].lock);

• Until TM interface solidifies...
• ... speculatively transactify lock-based programs in hardware
  - Speculative Lock Elision (SLE) [Rajwar+, MICRO’01]
    + No need to rewrite programs
    + Can always fall back on lock-based execution (overflow, I/O, etc.)
  - Modified rumor: this is what Sun’s Rock actually does

Roadmap Checkpoint

• Thread-level parallelism (TLP)
• Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing

• Synchronization
  - Lock implementation
  - Locking gotchas

• Cache coherence
  - Bus-based protocols
  - Directory protocols

• Memory consistency models
Recall: Simplest Multiprocessor

What if we don’t want to share the L1 caches?
- Bandwidth and latency issue
- Solution: use per-processor (“private”) caches
  - Coordinate them with a Cache Coherence Protocol

Shared-Memory Multiprocessors

Conceptual model
- The shared-memory abstraction
- Familiar and feels natural to programmers
- Life would be easy if systems actually looked like this...

Shared-Memory Multiprocessors

...but systems actually look more like this
- Processors have caches
- Memory may be physically distributed
- Arbitrary interconnect

Revisiting Our Motivating Example

Two $100 withdrawals from account #241 at two ATMs
- Each transaction maps to thread on different processor
- Track \texttt{accts[241].bal} (address is in $r3)
No-Cache, No-Problem

- Scenario I: processors have no caches
  - No problem

Write-Through Doesn't Fix It

- Scenario II(b): processors have write-through caches
  - This time only 2 (different) copies of \texttt{accts[241].bal}
  - No problem? What if another withdrawal happens on processor 0?

Cache Incoherence

- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of \texttt{accts[241].bal}: memory, p0$, p1$
  - Can get incoherent (inconsistent)

What To Do?

- No caches?
  - Slow
- Make shared data uncachable?
  - Faster, but still too slow
- Entire \texttt{accts} database is technically "shared"
  - Definition of "loosely shared"
- Data only really shared if two ATMs access same acct at once
- Flush all other caches on writes to shared data?
  - May as well not have caches

- Hardware cache coherence
  - Rough goal: all caches have same data at all times
  - Minimal flushing, maximum caching $\rightarrow$ best performance
Bus-based Multiprocessor

- Simple multiprocessors use a bus
  - All processors see all requests at the same time, same order
- Memory
  - Single memory module, -or-
  - Banked memory module

```
<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>
```

Bus

M_0 M_1 M_2 M_3

Hardware Cache Coherence

- Coherence
  - all copies have same data at all times
- Coherence controller:
  - Examines bus traffic (addresses and data)
  - Executes coherence protocol
    - What to do with local copy when you see different things happening on bus
- Three processor-initiated events
  - R: read W: write WB: write-back
- One response event: SD: send data
- Two remote-initiated events
  - BR: bus-read, read miss from another processor
  - BW: bus-write, write miss from another processor

VI (MI) Coherence Protocol

- **VI (valid-invalid) protocol**: aka MI
  - Two states (per block in cache)
    - V (valid): have block
    - I (invalid): don't have block
  + Can implement with valid bit
- Protocol diagram (left)
  - Convention: event→generated-event
  - Summary
    - If anyone wants to read/write block
    - Give it up: transition to I state
    - Write-back if your own copy is dirty
- This is an invalidate protocol
- Update protocol: copy data, don’t invalidate
  - Sounds good, but wastes a lot of bandwidth

VI Protocol State Transition Table

<table>
<thead>
<tr>
<th>State</th>
<th>This Processor</th>
<th>Other Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>Invalid (I)</td>
<td>Miss V</td>
<td>Miss V</td>
</tr>
<tr>
<td>Valid (V)</td>
<td>Hit</td>
<td>Hit</td>
</tr>
</tbody>
</table>

- Rows are “states”
  - I vs V
- Columns are “events”
  - Writeback events not shown
- Memory controller not shown
  - Responds with no other processor would respond
**VI Protocol (Write-Back Cache)**

- `lw` by processor 1 generates a BR (bus read)
- Processor 0 responds by sending its dirty copy, transitioning to `I`

**VI → MSI**

- VI protocol is inefficient
  - Only one cached copy allowed in entire system
  - Multiple copies can't exist even if read-only
    - Not a problem in example
    - Big problem in reality
- **MSI (modified-shared-invalid)**
  - Fixes problem: splits "V" state into two states
    - M (modified): local dirty copy
    - S (shared): local clean copy
  - Allows either
    - Multiple read-only copies (S-state) --OR--
    - Single read/write copy (M-state)

**MSI Protocol State Transition Table**

<table>
<thead>
<tr>
<th>State</th>
<th>Load</th>
<th>Store</th>
<th>Other Processor Load Miss</th>
<th>Store Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid (I)</td>
<td>Miss</td>
<td>M</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Hit</td>
<td>Upg Miss</td>
<td>---</td>
<td>I</td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Hit</td>
<td>Hit</td>
<td>Send Data S</td>
<td>I</td>
</tr>
</tbody>
</table>

- M → S transition also updates memory
  - After which memory will respond (as all processors will be in S)

**MSI Protocol (Write-Back Cache)**

- `lw` by processor 1 generates a BR
  - Processor 0 responds by sending its dirty copy, transitioning to S
- `sw` by processor 1 generates a BW
  - Processor 0 responds by transitioning to I
Cache Coherence and Cache Misses

- Coherence introduces two new kinds of cache misses
  - **Upgrade miss**: delay to acquire write permission to read-only block
  - **Coherence miss**: miss to a block evicted by bus event
- Example: direct-mapped 4B cache, 2B blocks

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Request</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>TT0B</td>
<td>1100 R</td>
<td>compulsory miss</td>
</tr>
<tr>
<td>TT1B</td>
<td>1100 W</td>
<td>upgrade miss</td>
</tr>
<tr>
<td>1100</td>
<td>1101:S</td>
<td>0010 BW</td>
</tr>
<tr>
<td>1100</td>
<td>1101:M</td>
<td>1101 BW</td>
</tr>
<tr>
<td>----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>1100</td>
<td>1101:S</td>
<td>0000 R</td>
</tr>
<tr>
<td>0000</td>
<td>0001:S</td>
<td>1100 W</td>
</tr>
</tbody>
</table>

**Exclusive Clean Protocol Optimization**

Processor 0
- 0: addi $r3,$r1,&accts
- 1: lw $r4,0($r3)
- 2: blt $r4,$r2,6
- 3: sub $r4,$r4,$r2
- 4: sw $r4,0($r3)
- 5: jal dispense_cash

Processor 1
- 0: addi $r3,$r1,&accts
- 1: lw $r4,0($r3)
- 2: blt $r4,$r2,6
- 3: sub $r4,$r4,$r2
- 4: sw $r4,0($r3)
- 5: jal dispense_cash

CPU0 CPU1 Mem
(No miss) 500 500
M:400 500
S:400 400

- Most modern protocols also include **E (exclusive)** state
  - Interpretation: “I have the only cached copy, and it’s a clean copy”
  - Why would this state be useful?

**MESI Protocol State Transition Table**

- **Load misses lead to “E” if no other processors is caching the block**

<table>
<thead>
<tr>
<th>State</th>
<th>This Processor</th>
<th>Other Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid (I)</td>
<td>Miss S or E</td>
<td>Miss M</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Hit</td>
<td>Upg Miss M</td>
</tr>
<tr>
<td>Exclusive (E)</td>
<td>Hit</td>
<td>Hit M</td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Hit</td>
<td>Hit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>This Processor</th>
<th>Other Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid (I)</td>
<td>Miss S or E</td>
<td>Miss M</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Hit</td>
<td>Upg Miss M</td>
</tr>
<tr>
<td>Exclusive (E)</td>
<td>Hit</td>
<td>Hit M</td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Hit</td>
<td>Hit</td>
</tr>
</tbody>
</table>

- Load misses lead to “E” if no other processors is caching the block
MESI Protocol and Cache Misses

- MESI protocol reduces upgrade misses
  - And writeback traffic

<table>
<thead>
<tr>
<th>Cache contents (prior to access)</th>
<th>Request</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>_____</td>
<td>_____</td>
<td>_____</td>
</tr>
<tr>
<td>1100</td>
<td>1101:E</td>
<td>_____</td>
</tr>
<tr>
<td>1100</td>
<td>1101:M</td>
<td>_____</td>
</tr>
<tr>
<td>1100</td>
<td>1101:M</td>
<td>_____</td>
</tr>
<tr>
<td>1100</td>
<td>1101:E</td>
<td>_____</td>
</tr>
<tr>
<td>0000</td>
<td>0001:S</td>
<td>_____</td>
</tr>
<tr>
<td>0000</td>
<td>0001:S</td>
<td>_____</td>
</tr>
</tbody>
</table>

Snooping Bandwidth Requirements

- Coherence events generated on...
  - L2 misses (and writebacks)
- Some parameters
  - 2 GHz CPUs, 2 IPC, 33% memory operations,
  - 2% of which miss in the L2, 64B blocks, 50% dirty
  - \((0.33 \times 0.02 \times 1.5) = 0.01\) events/insn
  - \(0.01\) events/insn \* 2 insn/cycle \* 2 cycle/ns = 0.04 events/ns
  - Address request: 0.04 events/ns \* 4 B/event = 0.16 GB/s
  - Data response: 0.04 events/ns \* 64 B/event = 2.56 GB/s
- That’s 2.5 GB/s ... per processor
  - With 16 processors, that’s 40 GB/s!
  - With 128 processors, that’s 320 GB/s!!
  - You can use multiple buses... but that hinders global ordering

More Snooping Bandwidth Problems

- Bus bandwidth is not the only problem
- Also **processor snooping bandwidth**
  - 0.01 events/insn \* 2 insn/cycle = 0.02 events/cycle per processor
  - 16 processors: 0.32 bus-side tag lookups per cycle
    - Add 1 port to cache tags? Sure
    - Invalidate over upgrade: Tags smaller data, ports less expensive
  - 128 processors: 2.56 bus-side tag lookups per cycle!
    - Add 3 ports to cache tags? Oy vey!
    - Implementing **inclusion** (L1 is strict subset of L2) helps a little
      - 2 additional ports on L2 tags only
      - Processor doesn’t use existing tag port most of the time
      - If L2 doesn’t care (99% of the time), no need to bother L1
        - Still kind of bad though
- **Upshot**: bus-based coherence doesn’t scale well

Scalable Cache Coherence

- **Part I: bus bandwidth**
  - Replace non-scalable bandwidth substrate (bus)...
    - ...with scalable one (point-to-point network, e.g., mesh)
- **Part II: processor snooping bandwidth**
  - Most snoops result in no action
  - Replace non-scalable broadcast protocol (spam everyone)...
    - ...with scalable **directory protocol** (only notify processors that care)
Scalable Cache Coherence

- Point-to-point interconnects
  - **Glueless MP**: no need for additional “glue” chips
  - Can be arbitrarily large: 1000’s of processors
    - **Massively parallel processors (MPPs)**
    - Only government (DoD) has MPPs...
    - Companies have much smaller systems: 32–64 processors
  - **Scalable multi-processors**
  - AMD Opteron/Phenom – point-to-point, glueless, broadcast
- Distributed memory: non-uniform memory architecture (NUMA)

Directory Coherence Protocols

- Observe: address space statically partitioned
  - Can easily determine which memory module holds a given line
  - That memory module sometimes called “home”
  - Can’t easily determine which processors have line in their caches
- Bus-based protocol: broadcast events to all processors/caches
  - Simple and fast, but non-scalable
- **Directories**: non-broadcast coherence protocol
  - Extend memory to track caching information
  - For each physical cache line whose home this is, track:
    - **Owner**: which processor has a dirty copy (I.e., M state)
    - **Sharers**: which processors have clean copies (I.e., S state)
  - Processor sends coherence event to home directory
  - Home directory only sends events to processors that care

MSI Directory Protocol

- Processor side
  - Directory follows its own protocol (obvious in principle)
- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
- Minus grayed out arcs/actions
  - Bus events that would not trigger action anyway
  - Directory won’t bother you unless you need to act

Directory MSI Protocol

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>P0</th>
<th>P1</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi r1,accts,r3</td>
<td>0: addi r1,accts,r3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>1: ld 0(r3),r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2: blt r4,r2,6</td>
<td>2: blt r4,r2,6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td>3: sub r4,r2,r4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>4: st r4,0(r3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5: call dispense_cash</td>
<td>5: call dispense_cash</td>
<td>M:400</td>
<td>M:400</td>
<td>(state)</td>
</tr>
</tbody>
</table>

- **1d** by P1 sends BR to directory
  - Directory sends BR to P0, P0 sends P1 data, does WB, goes to S
- **st** by P1 sends BW to directory
  - Directory sends BW to P0, P0 goes to I
Directory Flip Side: Latency

- Directory protocols
  + Lower bandwidth consumption → more scalable
    - Longer latencies
- Two read miss situations
  
  ![Diagram]

- Unshared: get data from memory
  - Snooping: 2 hops (P₀ → memory → P₀)
  - Directory: 2 hops (P₀ → memory → P₀)
- Shared or exclusive: get data from other processor (P₁)
  - Assume cache-to-cache transfer optimization
  - Snooping: 2 hops (P₀ → P₁ → P₀)
  - Directory: 3 hops (P₀ → memory → P₁ → P₀)
  - Common, with many processors high probability someone has it

Directory Flip Side: Complexity

- Latency not only issue for directories
  - Subtle correctness issues as well
    - Stem from unordered nature of underlying inter-connect
    - Individual requests to single cache must be ordered
      - Bus-based Snooping: all processors see all requests in same order
        - Ordering automatic
      - Point-to-point network: requests may arrive in different orders
        - Directory has to enforce ordering explicitly
        - Cannot initiate actions on request B...
        - Until all relevant processors have completed actions on request A
        - Requires directory to collect acks, queue requests, etc.
- Directory protocols
  - Obvious in principle
    - Complicated in practice

Coherence on Real Machines

- Many uniprocessors designed with on-chip snooping logic
  - Can be easily combined to form multi-processors
    - E.g., Intel Pentium4 Xeon
    - Multi-core
- Larger scale (directory) systems built from smaller MPs
  - E.g., Sun Wildfire, NUMA-Q, IBM Summit
- Some shared memory machines are **not cache coherent**
  - E.g., CRAY-T3D/E
  - Shared data is uncachable
  - If you want to cache shared data, copy it to private data section
  - Basically, cache coherence implemented in software
    - Have to really know what you are doing as a programmer

Best of Both Worlds?

- Ignore processor snooping bandwidth for a minute
- Can we combine best features of snooping and directories?
  - From snooping: fast two-hop cache-to-cache transfers
  - From directories: scalable point-to-point networks
  - In other words...
- Can we use broadcast on an unordered network?
  - Yes, and most of the time everything is fine
  - But sometimes it isn’t … **protocol race**
- Research Proposal: **Token Coherence (TC)**
  - An unordered broadcast snooping protocol … without data races
**Roadmap Checkpoint**

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- **Memory consistency models**

**Hiding Store Miss Latency**

- **Recall (back from caching unit)**
  - Hiding store miss latency
  - How? Write buffer
- Said it would complicate multiprocessors
  - Yes. It does.

**Recall: Write Misses and Write Buffers**

- **Read miss?**
  - Load can’t go on without the data, it must stall
- **Write miss?**
  - Technically, no instruction is waiting for data, why stall?

**Write buffer**: a small buffer

- Stores put address/value to write buffer, keep going
- Write buffer writes stores to D$ in the background
- Loads must search write buffer (in addition to D$)
  - Eliminates stalls on write misses (mostly)
  - Creates some problems (later)

**Write buffer vs. writeback-buffer**

- Write buffer: “in front” of D$, for hiding store misses
- Writeback buffer: “behind” D$, for hiding writebacks

**Memory Consistency**

- **Memory coherence**
  - Creates globally uniform (consistent) view...
  - Of a single memory location (in other words: cache line)
    - Not enough
      - Cache lines A and B can be individually consistent...
      - But inconsistent with respect to each other
- **Memory consistency**
  - Creates globally uniform (consistent) view...
  - Of all memory locations relative to each other
- Who cares? Programmers
  - Globally inconsistent memory creates mystifying behavior
Coherence vs. Consistency

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0’s write of flag before write of A!!! How?
  - Maybe coherence event of A is delayed somewhere in network
  - Or P0 has a coalescing write buffer that reorders writes

- Imagine trying to figure out why this code sometimes “works” and sometimes doesn’t
- **Real systems** act in this strange manner

Sequential Consistency (SC)

- **Formal definition of memory view programmers expect**
- Processors see their own loads and stores in program order + Provided naturally, even with out-of-order execution
- But also: processors see others’ loads and stores in program order
- And finally: all processors see same global load/store ordering – Last two conditions not naturally enforced by coherence

SC Doesn’t “Happen Naturally” Why?

- What is consistency concerned with?
  - P1 doesn’t actually view P0’s committed loads and stores
  - Views their **coherence events** instead
  - “Consistency model”: how observed order of coherence events relates to order of committed insns

- What does SC say?
  - Coherence event order must match committed insn order
    - And be identical for all processors
  - Let’s see what that implies

Enforcing SC

- **What does it take to enforce SC?**
  - Definition: all loads/stores globally ordered
  - Use ordering of coherence events to order all loads/stores

- **When do coherence events happen naturally?**
  - On cache access
  - For stores: retirement → in-order → good
    - No write buffer? Yikes, but OK with write-back D$
  - For loads: execution → out-of-order → bad
    - No out-of-order execution? Double yikes

- Is it true that multi-processors cannot be out-of-order?
  - That would be really bad
  - Out-of-order is needed to hide cache miss latency
  - And multi-processors not only have more misses...
    - ... but miss handling takes longer (coherence actions)
SC + Out-of-Order

- Recall: opportunistic load scheduling in a uni-processor
  - Loads issue speculatively relative to older stores
    - Stores scan for younger loads to same address have issued
    - Find one? Ordering violation → flush and restart
    - In-flight loads effectively “snoop” older stores from same process

- SC + OOO can be reconciled using same technique
  - "Invalidation" requests from other processors snoop in-flight loads
  - Think of load/store queue as extension of the cache hierarchy
  - MIPS R10K does this

- SC implementable, but overheads still remain:
  - Write buffer issues
  - Complicated load/store logic

What is this?
- P0 sees P1’s write of A
- P0 should also see P1’s write of B (older than write of A)
  - But doesn’t because it read of B out-of-order w.r.t. read of A
- Does this mean no out-of-order? (that would be bad)
- Fortunately, there is a way
SC + Write Buffers

- Store misses are slow
  - Global acquisition of M state (write permission)
  - Multiprocessors have more store misses than uniprocessors
    - Upgrade miss: I have block in S, require global upgrade to M

- Apparent solution: write buffer
  - Commit store to write buffer, let it absorb store miss latency
  - But a write buffer means...
    - I see my own stores commit before everyone else sees them

- Orthogonal to out-of-order execution
  - Even in-order processors have write buffers

### Is SC Really Necessary?

- SC
  - Most closely matches programmer’s intuition (don’t under-estimate)
    - Restricts optimization by compiler, CPU, memory system
    - Supported by MIPS, HP PA-RISC

- Is full-blown SC really necessary? What about...
  - All processors see others’ loads/stores in program order
    - But not all processors have to see same global order
  + Allows processors to have in-order write buffers
    - Doesn’t confuse programmers too much
      - Synchronized programs (e.g., our example) work as expected

- Processor Consistency (PC): e.g., Intel/AMD x86, SPARC

### SC + Write Buffers

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=0; B=0;</td>
<td></td>
</tr>
<tr>
<td>A=1;</td>
<td>B=1;</td>
</tr>
<tr>
<td>if(B==0) // in-order commit</td>
<td>if(A==0) // in-order commit</td>
</tr>
<tr>
<td>A=1;</td>
<td>B=1;</td>
</tr>
<tr>
<td>// in-order to D$</td>
<td>// in-order to D$</td>
</tr>
</tbody>
</table>

- Possible for both (B==0) and (A==0) to be true
- Because B=1 and A=1 are just sitting in the write buffers
  - Which is wrong
  - So does SC mean no write buffer?
    - Yup, and that hurts

- Research direction: use deep speculation to hide latency
  - Beyond the out-of-order window, looks like transactional memory...

### Weak Memory Ordering

- For properly synchronized programs...
  - ...only acquire/release must be strictly ordered

- Why? acquire-release pairs define critical sections
  - Between critical-sections: data is private
    - Globally unordered access OK
  - Within critical-section: access to shared data is exclusive
    - Globally unordered access also OK

- Implication: compiler or dynamic scheduling is OK
  - As long as re-orderings do not cross synchronization points

- Weak Ordering (WO): Alpha, Itanium, ARM, PowerPC
  - ISA provides fence fence to indicate scheduling barriers
  - Proper use of fences is somewhat subtle
  + Use synchronization library, don’t write your own
Pop Quiz!

- Answer the following questions:
  - Initially: all variables zero (that is, x is 0, y is 0, flag is 0, A is 0)
  - What value pairs can be read by the two loads? (x, y) pairs:
    
    | thread 1 | thread 2 |
    |-----------|----------|
    | ld x      | st 1 → y|
    | ld y      | st 1 → x|
  
  - What value pairs can be read by the two loads? (x, y) pairs:
    
    | thread 1 | thread 2 |
    |-----------|----------|
    | st 1 → y | st 1 → x|
    | ld x      | ld y     |
  
  - What value can be read by the load A?
    
    | thread 1 | thread 2 |
    |-----------|----------|
    | st 1 → A | while(flag == 0) { }|
    | st 1 → flag | ld A |

Fences aka Memory Barriers

- **Fences (memory barriers):** special insns
  - Ensure that loads/stores don’t cross acquire release boundaries
  - Very roughly
    - acquire
    - fence
    - critical section
    - fence
    - release
  
  - How do they work?
    - fence insn must commit before any younger insn dispatches
      - This also means write buffer is emptied
        - Makes lock acquisition and release slow(er)
  
  - Use synchronization library, don’t write your own

Multiprocessing & Power Consumption

- Multiprocessing can be very power efficient
- Recall: dynamic voltage and frequency scaling
  - Performance vs power is NOT linear
  - Example: Intel’s Xscale
    - 1 GHz → 200 MHz reduces energy used by 30x
- Impact of parallel execution
  - What if we used 5 Xscales at 200Mhz?
    - Similar performance as a 1Ghz Xscale, but **1/6th the energy**
  - 5 cores * 1/30th = 1/6th
- Assumes parallel speedup (a difficult task)
  - Remember Ahmdal’s law

Shared Memory Summary

- **Synchronization:** regulated access to shared data
  - Key feature: atomic lock acquisition operation (e.g., t&s)
  - Performance optimizations: test-and-test-and-set, queue locks
- **Coherence:** consistent view of individual cache lines
  - Absolute coherence not needed, relative coherence OK
  - VI and MSI protocols, cache-to-cache transfer optimization
  - Implementation? snooping, directories
- **Consistency:** consistent view of all memory locations
  - Programmers intuitively expect sequential consistency (SC)
    - Global interleaving of individual processor access streams
      - Not always naturally provided, may prevent optimizations
  - Weaker ordering: consistency only for synchronization points
Summary

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Synchronization
  - Lock implementation
  - Locking gotchas
- Cache coherence
  - Bus-based protocols
  - Directory protocols
- Memory consistency models