CIS 371
Computer Organization and Design

Unit 11: Multicore

Slides developed by M. Martin, A. Roth & C.J. Taylor at the University of Pennsylvania with sources that included University of Wisconsin slides by Mark Hill, Guri Sohi, Jim Smith, and David Wood
This Unit: Shared Memory Multiprocessors

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- Memory consistency models
Readings

• P&H
  • Chapter 7.1-7.3, 7.5
  • Chapter 5.8, 5.10
Beyond Implicit Parallelism

- Consider “daxpy”:
  ```c
  double a, x[SIZE], y[SIZE], z[SIZE];
  void daxpy():
      for (i = 0; i < SIZE; i++)
          z[i] = a*x[i] + y[i];
  ```

- Lots of instruction-level parallelism (ILP)
  - Great!
  - But how much can we really exploit? 4 wide? 8 wide?
    - Limits to (efficient) super-scalar execution

- But, if SIZE is 10,000, the loop has 10,000-way parallelism!
  - How do we exploit it?
Explicit Parallelism

• Consider “daxpy”:
  
  ```
  double a, x[SIZE], y[SIZE], z[SIZE];
  void daxpy():
      for (i = 0; i < SIZE; i++)
          z[i] = a*x[i] + y[i];
  ```

• Break it up into N “chunks” on N cores!
  • Done by the programmer (or maybe a really smart compiler)

  ```
  void daxpy(int chunk_id):
      chuck_size = SIZE / N
      my_start = chunk_id * chuck_size
      my_end = my_start + chuck_size
      for (i = my_start; i < my_end; i++)
          z[i] = a*x[i] + y[i]
  ```

• Assumes
  • Local variables are “private” and x, y, and z are “shared”
  • Assumes SIZE is a multiple of N (that is, SIZE % N == 0)

<table>
<thead>
<tr>
<th>Chunk ID</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>99</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>199</td>
</tr>
<tr>
<td>2</td>
<td>200</td>
<td>299</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>399</td>
</tr>
</tbody>
</table>

SIZE = 400, N=4
Explicit Parallelism

• Consider “daxpy”:

```c
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy(int chunk_id):
    chuck_size = SIZE / N
    my_start = chunk_id * chuck_size
    my_end = my_start + chuck_size
    for (i = my_start; i < my_end; i++)
        z[i] = a*x[i] + y[i]
```

• Main code then looks like:

```c
parallel_daxpy():
    for (tid = 0; tid < CORES; tid++) {
        spawn_task(daxpy, tid);
    }
    wait_for_tasks(CORES);
```
Explicit (Loop-Level) Parallelism

• Another way: “OpenMP” annotations to inform the compiler

```c
double a, x[SIZE], y[SIZE], z[SIZE];
void daxpy() {
    #pragma omp parallel for
    for (i = 0; i < SIZE; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

• But only works if loop is actually parallel
  • If not parallel, incorrect behavior may result in unpredictable ways
Multicore & Multiprocessor Hardware
Multiplying Performance

- A single core can only be so fast
  - Limited clock frequency
  - Limited instruction-level parallelism

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

- Old-school (2000s): Ultra Enterprise 25k
  - 72 dual-core UltraSPARC IV+ processors
  - Up to 1TB of memory
  - Niche: large database servers
  - $$\$, weights more than 1 ton

- Today: multicore is everywhere
  - Dual-core ARM phones
Intel Quad-Core “Core i7”
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 i7 Quad**
    - Four cores, private L2s
    - Shared 8 MB L3
  - **Sun Niagara**
    - 8 cores, each 4-way threaded
    - Shared 2MB L2
    - For servers, not desktop

Why multicore? What else would you do with 1 billion transistors?
Sun Niagara II
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*
Amdahl’s Law

- Restatement of the law of diminishing returns
  - Total speedup limited by non-accelerated piece
  - Analogy: drive to work & park car, walk to building

- Consider a task with a “parallel” and “serial” portion
  - What is the speedup with N cores?
  - Speedup(n, p, s) = (s+p) / (s + (p/n))
    - p is “parallel percentage”, s is “serial percentage”
  - What about infinite cores?
    - Speedup(p, s) = (s+p) / s = 1 / s

- Example: can optimize 50% of program A
  - Even “magic” optimization that makes this 50% disappear...
  - ...only yields a 2X speedup
Amdahl’s Law Graph

“Threading” & The Shared Memory Execution Model
First, Uniprocessor Concurrency

- **Software “thread”:** Independent flows of execution
  - “Per-thread” state
    - Context state: PC, registers
    - Stack (per-thread local variables)
  - “Shared” state: globals, heap, etc.
  - Threads generally share the same memory space
    - “Process” like a thread, but different memory space
  - Java has thread support built in, C/C++ using a thread library
- Generally, system software (the O.S.) manages threads
  - “Thread scheduling”, “context switching”
  - In single-core system, all threads share the one processor
    - Hardware timer interrupt occasionally triggers O.S.
    - Quickly swapping threads gives illusion of concurrent execution
  - Much more in an operating systems course
Multithreaded Programming Model

- Programmer explicitly creates multiple threads
- All loads & stores to a single *shared memory* space
  - Each thread has its own stack frame for local variables
  - All memory shared, accessible by all threads
- 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)**
    - This is our focus!
Shared Memory Model: Interleaving

- **Initially: all variables zero** (that is, x is 0, y is 0)

  - **thread 1**
    - store 1 → y
    - load x

  - **thread 2**
    - store 1 → x
    - load y

- What value pairs can be read by the two loads?
Shared Memory Model: Interleaving

- **Initially:** all variables zero (that is, x is 0, y is 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 → y</td>
<td>store 1 → x</td>
</tr>
<tr>
<td>load x</td>
<td>load y</td>
</tr>
</tbody>
</table>

- What value pairs can be read by the two loads?

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 → y</td>
<td>store 1 → y</td>
</tr>
<tr>
<td>load x</td>
<td>load y</td>
</tr>
<tr>
<td>store 1 → x</td>
<td>store 1 → x</td>
</tr>
<tr>
<td>load x</td>
<td>load x</td>
</tr>
<tr>
<td>load y</td>
<td>load y</td>
</tr>
<tr>
<td>store 1 → x</td>
<td>store 1 → x</td>
</tr>
<tr>
<td>load x</td>
<td>load x</td>
</tr>
<tr>
<td>store 1 → y</td>
<td>store 1 → y</td>
</tr>
<tr>
<td>load y</td>
<td>load y</td>
</tr>
<tr>
<td>store 1 → y</td>
<td>store 1 → y</td>
</tr>
<tr>
<td>load y</td>
<td>load y</td>
</tr>
</tbody>
</table>

- What about (x=0, y=0)?
Shared Memory Implementations

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

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

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

- **All support the shared memory programming model**
Simplest Multiprocessor

- Replicate entire processor pipeline!
  - Instead of replicating just register file & PC
  - Exception: share the caches (we’ll address this bottleneck soon)

- Multiple threads execute
  - Shared memory programming model
  - Operations (loads and stores) are interleaved “at random”
  - Loads returns the value written by most recent store to location
Hardware Multithreading

- **Hardware Multithreading (MT)**
  - Multiple threads dynamically share a single pipeline
  - Replicate only per-thread structures: program counter & registers
  - Hardware interleaves instructions

  + **Multithreading improves utilization and throughput**
    - Single programs utilize <50% of pipeline (branch, cache miss)
  + **Multithreading does not improve single-thread performance**
    - Individual threads run as fast or even slower
  + **Coarse-grain MT**: switch on cache misses Why?
  + **Simultaneous MT**: no explicit switching, fine-grain interleaving
Four Shared Memory Issues

1. **Cache coherence**
   - If cores have private (non-shared) caches
   - How to make writes to one cache “show up” in others?

2. **Parallel programming**
   - How does the programmer express the parallelism?

3. **Synchronization**
   - How to regulate access to shared data?
   - How to implement “locks”?

4. **Memory consistency models**
   - How to keep programmer sane while letting hardware optimize?
   - How to reconcile shared memory with compiler optimizations, store buffers, and out-of-order execution?
Roadmap Checkpoint

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- 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**

- Must still provide shared-memory invariant:
  - “**Loads read the value written by the most recent store**”
No-Cache (Conceptual) Implementation

Memory

P₀

P₁

P₂
No-Cache (Conceptual) Implementation

- No caches
  - Not a realistic design

<table>
<thead>
<tr>
<th>Memory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
Shared Cache Implementation

- On-chip shared cache
- Lacks per-core caches
  - Shared cache becomes bottleneck

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>A</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
Shared Cache Implementation

1. Load [A] from memory.
Shared Cache Implementation

1. Load [A] (500)
2. Memory
3. Shared Cache
4. Interconnect

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interconnect

Shared Cache

Tag: A
Data: 500

Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
Shared Cache Implementation

- Write into cache

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
</tr>
</tbody>
</table>

Memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>

Interconnect

Store 400 -> [A]
Shared Cache Implementation

Store 400 -> [A]

- Mark as “dirty”
- Memory not updated
Adding Private Caches

- **Add per-core caches** (write-back caches)
  - Reduces latency
  - Increases throughput
  - Decreases energy

Interconnect

<table>
<thead>
<tr>
<th>Cache</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared Cache</th>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Adding Private Caches

1. Load [A]

2. Interconnect

3. Memory

Cache
<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
</table>

Cache
<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
</table>

Cache
<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
</table>

Interconnect

Shared Cache
<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
</table>

Memory
<table>
<thead>
<tr>
<th>A</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
Adding Private Caches

1. Load [A] (500)
2. Memory
3. Shared Cache
4. Interconnect
5. Cache
6. Cache
Adding Private Caches

1. Store 400 -> [A]

- Cache P0
  - Tag
  - Data

- Cache P1
  - Tag
  - Data
  - A 400

- Cache P2
  - Tag
  - Data

- Shared Cache
  - Tag
  - Data
  - State
  - A 500
    - Clean

- Memory
  - A 500
  - B
Adding Private Caches

- **P₀** Cache
  - **Cache**
  - **Tag**
  - **Data**

- **P₁** Cache
  - **Cache**
  - **Tag**: A
  - **Data**: 400
  - **State**: Dirty
  - **Store 400 -> [A]**

- **P₂** Cache
  - **Cache**
  - **Tag**
  - **Data**

- **Interconnect**

- **Shared Cache**
  - **Tag**: A
  - **Data**: 500
  - **State**: Clean

- **Memory**
  - **Tag**: A
  - **Data**: 500
  - **Tag**: B
Private Cache Problem: Incoherence

- What happens with another core tries to read A?
Private Cache Problem: Incoherence

1. **Load [A]**

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
</table>

P₀

State: Dirty

P₁

State: Clean

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
<td>Clean</td>
</tr>
</tbody>
</table>

Interconnect

Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
Private Cache Problem: Incoherence

1. Load [A] (500)
2. Memory
3. Shared Cache
4. Interconnect

- Cache:
  - P0: Tag A, Data 500
  - P1: Tag A, Data 400, State Dirty
  - P2

- Shared Cache:
  - Tag A, Data 500, State Clean

- Memory:
  - A 500
  - B
Private Cache Problem: Incoherence

- \( P_0 \) got the wrong value!
Rewind: Fix Problem by Tracking Sharers

- **Solution**: Track copies of each block

### Memory

- A: 500
- B

### Shared Cache

- A: 500

### Shared Cache Sharers

- P1

### Cache

- **P0**
  - Tag | Data | State
  - A

- **P1**
  - Tag | Data | State
  - A: 400 | Dirty

- **P2**
  - Tag | Data | State
  - A

### Interconnect
Use Tracking Information to “Invalidate”

1. Load [A]

2. Shared Cache

Memory

Interconnect
Use Tracking Information to “Invalidate”

1. Load [A]
2. Share Cache
3. Share [A] with P1

Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
<td>--</td>
<td>P1</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>Dirty</td>
</tr>
</tbody>
</table>

Interconnect

P₀

P₁

P₂
Use Tracking Information to “Invalidate”

1. Load [A] (400)
2. Share cache
3. Memory
4. Interconnect
5. Cache
“Valid/Invalid” Cache Coherence

• To enforce the shared memory invariant...
  • “Loads read the value written by the most recent store”

• Enforce the invariant...
  • “At most one valid copy of the block”
  • Simplest form is a two-state “valid/invalid” protocol
  • If a core wants a copy, must find and “invalidate” it

• On a cache miss, how is the valid copy found?
  • Option #1 “Snooping”: broadcast to all, whoever has it responds
  • Option #2: “Directory”: track sharers at known location

• Problem: multiple copies can’t exist, even if read-only
  • Consider mostly-read data structures, instructions, etc.
MSI Cache Coherence Protocol

• Solution: enforce the invariant...
  • **Multiple read-only copies** —OR—
  • **Single read/write copy**

• Track these MSI permissions (states) in per-core caches
  • **Modified (M): read/write permission**
  • **Shared (S): read-only permission**
  • **Invalid (I): no permission**

• Also track a **“Sharer” bit vector** in shared cache
  • One bit per core; tracks all shared copies of a block
  • Then, *invalidate all readers* when a write occurs

• Allows for many readers...
  • ...while still enforcing shared memory invariant
    ("Loads read the value written by the most recent store")
Load [A]

Miss!

P0

P1

P2

Cache

Tag | Data | State
--- | --- | ---
-- | -- | --
-- | -- | --

Cache

Tag | Data | State
--- | --- | ---
A | 400 | M
-- | -- | --

Cache

Tag | Data | State
--- | --- | ---
-- | -- | --
-- | -- | --

Shared Cache

Tag | Data | State | Sharers
--- | --- | --- | ---
A | 500 | P1 is Modified | P1
B | 0 | Idle | --

Memory

Tag | Data
--- | ---
A | 500
B | 0

Point-to-Point Interconnect
MSI Coherence Example: Step #2

Load [A]

LdMiss: Addr=A

LdMissForward: Addr=A, Req=P0

P0

P1

P2

Cache

Tag | Data | State
---|------|------
-- | -- | --
-- | -- | --
A  | 400 | M
-- | -- | --

Cache

Tag | Data | State
---|------|------
-- | -- | --
-- | -- | --

Shared Cache

Tag | Data | State | Sharers
---|------|------|------
A  | 500  | Blocked | P1
B  | 0    | Idle   | --

Memory

A  | 500
B  | 0
MSI Coherence Example: Step #3

Load [A] to P₀

P₀

Cache

Tag | Data | State
--- | --- | ---
-- | -- | --
-- | -- | --

P₁

Cache

Tag | Data | State
--- | --- | ---
A | 400 | S
-- | -- | --

P₂

Cache

Tag | Data | State
--- | --- | ---
-- | -- | --
-- | -- | --

Shared Cache

Tag | Data | State | Sharers
--- | --- | --- | ---
A | 500 | Blocked | P₁
B | 0 | Idle | --

Memory

A | 500
B | 0

Point-to-Point Interconnect

Response: Addr=A, Data=400
MSI Coherence Example: Step #4

Point-to-Point Interconnect

Load [A]

Response: Addr=A, Data=400

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
<td>Blocked</td>
<td>P1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Idle</td>
<td>--</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
MSI Coherence Example: Step #5

1. Load [A] (400)

2. Unblock: Addr=A, Data=400

3. Point-to-Point Interconnect

4. Updated Cache States:
   - P0: Tag A, Data 400, State S
   - P1: Tag A, Data 400, State S
   - P2: Tag --, Data --, State --

5. Shared Cache State:
   - Tag A, Data 400, State Shared, Dirty, Sharers P0, P1
   - Tag B, Data 0, State Idle

6. Memory States:
   - A: 500
   - B: 0
MSI Coherence Example: Step #6

Point-to-Point Interconnect

P₀

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Miss!

P₁

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

P₂

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Store 300 -> [A]

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>Shared, Dirty</td>
<td>P₀, P₁</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Idle</td>
<td>--</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
MSI Coherence Example: Step #7

Store 300 -> [A]

UpgradeMiss: Addr=A

Point-to-Point Interconnect

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>Blocked</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Idle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
MSI Coherence Example: Step #8

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>--</td>
<td>I</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

Point-to-Point Interconnect

Store 300 -> [A]

Shared Cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>Blocked</td>
<td>P0, P1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Idle</td>
<td>--</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>

Invalidate: Addr=A, Req=P0, Acks=1
MSI Coherence Example: Step #9

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>S</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>--</td>
<td>I</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

**Point-to-Point Interconnect**

**Shared Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>Blocked</td>
<td>P0, P1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Idle</td>
<td>--</td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
MSI Coherence Example: Step #10

**Step 2:** Invalidate: Addr=A, Req=P0, Acks=1

**Step 3:** Ack: Addr=A, Acks=1

Store 300 -> [A]

CIS 371: Comp. Org.  |  Dr. Benedict J. Brown
MSI Coherence Example: Step #11

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>300</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>--</td>
<td>I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Shared Cache**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
<th>State</th>
<th>Sharers</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td><strong>P0 is Modified</strong></td>
<td>P0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>Idle</td>
<td>--</td>
</tr>
</tbody>
</table>

**Unblock: Addr=A**

Point-to-Point Interconnect

**Store 300 -> [A]**

**Memory**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>500</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>
MESI Cache Coherence

• Ok, we have read-only and read/write with MSI

• But consider load & then store of a block by same core
  • Under coherence as described, this would be two misses: “Load miss” plus an “upgrade miss”...
  • ... even if the block isn’t shared!
  • Consider programs with 99% (or 100%) private data
    • Potentially doubling number of misses (bad)

• Solution:
  • Most modern protocols also include E (exclusive) state
  • Interpretation: “I have the only cached copy, and it’s a clean copy”
    • Has read/write permissions
    • Just like “Modified” but “clean” instead of “dirty”.
MESI Operation

- **Goals:**
  - Avoid “upgrade” misses for non-shared blocks
  - While not increasing eviction (aka writeback or replacement) traffic

- **Two cases on a load miss to a block...**
  - **Case #1:** ... with no current sharers
    (that is, no sharers in the set of sharers)
    - Grant requester “Exclusive” copy with read/write permission
  - **Case #2:** ... with other sharers
    - As before, grant just a “Shared” copy with read-only permission

- A store to a block in “Exclusive” changes it to “Modified”
  - Instantaneously & silently (no latency or traffic)

- On block eviction (aka writeback or replacement)...
  - If “Modified”, block is dirty, must be written back to next level
  - If “Exclusive”, writing back the data is not necessary
    (but notification may or may not be, depending on the system)
Cache Coherence and Cache Misses

• With the “Exclusive” state...
  • Coherence has no overhead on misses to non-shared blocks
  • Just request/response like a normal cache miss
• But, coherence introduces two new kinds of cache misses
  • Upgrade miss: stores to read-only blocks
    • Delay to acquire write permission to read-only block
  • Coherence miss
    • Miss to a block evicted by another processor’s requests
• Making the cache larger...
  • Doesn’t reduce these types of misses
  • So, as cache grows large, these sorts of misses dominate
• False sharing
  • Two or more processors sharing parts of the same block
  • But not the same bytes within that block (no actual sharing)
  • Creates pathological “ping-pong” behavior
  • Careful data placement may help, but is difficult
Cache Coherence Protocols

- Two general types
  - Update-based cache coherence
    - Write through update to all caches
    - Too much traffic; used in the past, not common today
  - Invalidation-based cache coherence (examples shown)
- Of invalidation-based cache coherence, two types:
  - Snooping/broadcast-based cache coherence
    - No explicit state, but too much traffic; not common today
  - Directory-based cache coherence (examples shown)
    - Track sharers of blocks
- For directory-based cache coherence, two options:
  - Enforce “inclusion”; if in per-core cache, must be in last-level cache
  - Encoding sharers in cache tags (examples shown & Core i7)
  - No inclusion? “directory cache” parallel to last-level cache (AMD)
Scaling Cache Coherence

- **Scalable interconnect**
  - Build switched interconnect to communicate among cores

- **Scalable directory lookup bandwidth**
  - Address interleave (or “bank”) the last-level cache
  - Low-order bits of block address select which cache bank to access
  - Coherence controller per bank

- **Scalable traffic**
  - Amortized analysis shows traffic overhead independent of core #
  - Each invalidation can be tied back to some earlier request

- **Scalable storage**
  - Bit vector requires n-bits for n cores, scales up to maybe 32 cores
  - Inexact & “coarse” encodings trade more traffic for less storage

- Hierarchical design can help all of the above, too
- See: “Why On-Chip Cache Coherence is Here to Stay”, CACM, 2012
Coherence Recap & Alternatives

- Keeps caches “coherent”
  - Load returns the most recent stored value by any processor
  - And thus keeps caches transparent to software

- Alternatives to cache coherence
  - #1: no caching of shared data (slow)
  - #2: requiring software to explicitly “flush” data (hard to use)
    - Using some new instructions
  - #3: message passing (programming without shared memory)
    - Used in clusters of machines for high-performance computing

- However, directory-based coherence protocol scales well
  - Perhaps to 1000s of cores
Roadmap Checkpoint

- **Thread-level parallelism (TLP)**
- **Shared memory model**
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- **Cache coherence**
  - Valid/Invalid, MSI, MESI
- **Parallel programming**
- **Synchronization**
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- **Memory consistency models**
Parallel Programming
Parallel Programming

• One use of multiprocessors: **multiprogramming**
  • Running multiple programs with no interaction between them
  • Works great for a few cores, but what next?

• Or, programmers must **explicitly** express parallelism
  • “Coarse” parallelism beyond what the hardware can extract **implicitly**
  • Even the compiler can’t extract it in most cases

• How? Several options:
  1. Call libraries that perform well-known computations in parallel
     • Example: a matrix multiply routine, etc.
  2. Add code annotations ("this loop is parallel"), OpenMP
  3. Parallel “for” loops, task-based parallelism, ...
  4. Explicitly spawn “tasks”, runtime/OS schedules them on the cores

• Parallel programming: key challenge in multicore revolution
Example #1: Parallelizing Matrix Multiply

- How to parallelize matrix multiply?
  - Replace outer “for” loop with “parallel_for” or OpenMP annotation
  - Supported by many parallel programming environments

- Implementation: give each of N processors loop iterations
  ```c
  int start = (SIZE/N) * my_id(); // my_id() from library
  for (I = start; I < start + SIZE/N; I++)
    for (J = 0; J < SIZE/N; J++)
      for (K = 0; K < SIZE; K++)
  ```

- Each processor runs copy of loop above
  - No explicit synchronization required (implicit at end of loop)
Example #2: Bank Accounts

• Consider

```c
struct acct_t { int balance; … };  
struct acct_t accounts[MAX_ACCT];   // current balances

struct trans_t { int id; int amount; };  
struct trans_t transactions[MAX_TRANS]; // debit amounts

for (i = 0; i < MAX_TRANS; i++) {
    debit(transactions[i].id, transactions[i].amount);
}

void debit(int id, int amount) {
    if (accounts[id].balance >= amount) {
        accounts[id].balance -= amount;
    }
}
```

• Can we do these “debit” operations in parallel?
  • Does the order matter?
Example #2: Bank Accounts

Example of **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 global and thus **shared**, can’t register allocate
- `id` and `amt` are private variables, register allocated to `r1`, `r2`

- Running example

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

```assembly
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```
An Example Execution

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

```assembly
Thread 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)

Thread 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)
```
A Problem Execution

Thread 0
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
<<< Thread Switch >>>
4: st r4,0(r3)

Thread 1
0: addi r1,accts,r3
1: ld 0(r3),r4
2: blt r4,r2,done
3: sub r4,r2,r4
4: st r4,0(r3)

Mem
500

Time
500
400

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

- **Synchronization**: a key issue for shared memory
- Regulate access to shared data (mutual exclusion)
- Low-level primitive: **lock** (higher-level: “semaphore” or “mutex”)
  - Operations: `acquire(lock)` and `release(lock)`
  - Region between `acquire` and `release` is a **critical section**
  - Must interleave `acquire` and `release`
  - Interfering `acquire` will block
- Another option: **Barrier synchronization**
  - Blocks until all threads reach barrier, used at end of “parallel_for”

```c
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
shared int lock;
void debit(int id, int amt):
  acquire(lock);
  if (accts[id].bal >= amt) {
    accts[id].bal -= amt;
  }
  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>(still in acquire)</td>
</tr>
<tr>
<td>1: ld 0(r3),r4</td>
<td>0: addi r1,accts,r3</td>
</tr>
<tr>
<td>2: blt r4,r2,done</td>
<td>1: ld 0(r3),r4</td>
</tr>
<tr>
<td>3: sub r4,r2,r4</td>
<td>2: blt r4,r2,done</td>
</tr>
<tr>
<td>&lt;&lt;&lt; Switch &gt;&gt;&gt;</td>
<td>&lt;&lt;&lt; Switch &gt;&gt;&gt;</td>
</tr>
<tr>
<td>4: st r4,0(r3)</td>
<td>(still in acquire)</td>
</tr>
<tr>
<td>call release(lock)</td>
<td>0: addi r1,accts,r3</td>
</tr>
<tr>
<td></td>
<td>1: ld 0(r3),r4</td>
</tr>
<tr>
<td></td>
<td>2: blt r4,r2,done</td>
</tr>
<tr>
<td></td>
<td>3: sub r4,r2,r4</td>
</tr>
<tr>
<td></td>
<td>4: st r4,0(r3)</td>
</tr>
</tbody>
</table>

- Fixed, but how do we implement acquire & release?

Mem

500

Time

300

400

Mem

500
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
      
      A0:  ld 0(&lock),r6  
      A1:  bnez r6,A0  
      A2:  addi r6,1,r6  
      A3:  st r6,0(&lock)  

  - release(lock): lock = 0;
    
    R0:  st r0,0(&lock)       // r0 holds 0
Incorrect Lock Implementation

• 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
Correct Spin Lock: Use Atomic Swap

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

- New acquire sequence
  - (value of r1 is 1)
    - A0: `swap r1,0(&lock)`
    - A1: `bnez r1,A0`
  - If lock was initially busy (1), doesn’t change it, **keep looping**
  - 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
RISC Test-And-Set

- **swap**: a load and store in one insn is not very “RISC”
  - Broken up into micro-ops, but then how is it made atomic?
- “Load-link” / “store-conditional” pairs
  - Atomic load/store pair
    
    ```
    label:
    load-link r1,0(&lock) // potentially other insns
    store-conditional r2,0(&lock)
    branch-not-zero label  // check for failure
    ```
  - On **load-link**, processor remembers address...
    - ...And looks for writes by other processors
    - If write is detected, next **store-conditional** will fail
      - Sets failure condition
- Used by ARM, PowerPC, MIPS, Itanium
Lock Correctness

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: swap r1,0(&amp;lock)</td>
<td>A0: swap r1,0(&amp;lock)</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A0: swap r1,0(&amp;lock)</td>
</tr>
<tr>
<td>CRITICAL_SECTION</td>
<td>A1: bnez r1,#A0</td>
</tr>
<tr>
<td>A0: swap r1,0(&amp;lock)</td>
<td>A1: bnez r1,#A0</td>
</tr>
</tbody>
</table>

+ Lock actually works...
  - Thread 1 keeps spinning

- Sometimes called a “test-and-set lock”
  - Named after the common “test-and-set” atomic instruction
"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 swap, each of which includes a st
      - Repeated stores by multiple processors costly
      - Generating a ton of useless interconnect traffic

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0: swap r1,0(&amp;lock)</td>
<td>A0: swap r1,0(&amp;lock)</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A1: bnez r1,#A0</td>
</tr>
<tr>
<td>A0: swap r1,0(&amp;lock)</td>
<td>A1: bnez r1,#A0</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A0: swap r1,0(&amp;lock)</td>
</tr>
<tr>
<td>A1: bnez r1,#A0</td>
<td>A1: bnez r1,#A0</td>
</tr>
</tbody>
</table>
Test-and-Test-and-Set Locks

- **Solution:** **test-and-test-and-set locks**
  - New acquire sequence
    - A0: `ld r1,0(&lock)`
    - A1: `bnez r1,A0`
    - A2: `addi r1,1,r1`
    - A3: `swap r1,0(&lock)`
    - A4: `bnez r1,A0`
  - Within each loop iteration, before doing a **swap**
    - Spin doing a simple test (`ld`) to see if lock value has changed
    - Only do a **swap** (**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 swap requests

- **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 is descheduled by O.S.
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];  
shared Lock_t lock;  
void debit(int id, int amt) {  
    acquire(lock);  
    if (accts[id].bal >= amt) {  
        accts[id].bal -= amt;  
    }  
    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

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

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

- What about critical sections that require two locks?
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_t lock; ...};
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int 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

Thread 0

```plaintext
id_from = 241;
id_to = 37;

acquire(accts[241].lock);
// wait to acquire lock 37
// waiting...
// still waiting...
```

Thread 1

```plaintext
id_from = 37;
id_to = 241;

acquire(accts[37].lock);
// wait to acquire lock 241
// waiting...
// ...
```
Deadlock!
Multiple Locks And Deadlock

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
// waiting...
// ...

• **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 ...
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_t lock; ... }
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int 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

**Thread 0**

```c
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);
```

**Thread 1**

```c
id_from = 37;
id_to = 241;
id_first = min(37,241)=37;
id_second = max(37,241)=241;
// wait to acquire lock 37
// waiting…
// …
acquire(accts[37].lock);
```

- Great, are we done? No
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 un-necessary**
  - 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”
  - Programmer uses construct: “atomic { ... code ... }”
    - Hardware, compiler & runtime executes the code “atomically”
    - Uses speculation, rolls back on conflicting accesses
Research: Transactional Memory (TM)

- **Transactional Memory (TM) goals:**
  - Programming simplicity of coarse-grain locks
  - Higher concurrency (parallelism) of fine-grain locks
    - Critical sections only serialized if data is actually shared
  - Lower overhead than lock acquisition
  - Hot academic & industrial research topic (or was a few years ago)
  - No fewer than nine research projects:
    - Brown, Stanford, MIT, Wisconsin, Texas, Rochester, Sun/Oracle, Intel
    - Penn, too

- Most recently:
  - Intel announced TM support in “Haswell” core! (shipping in 2013)
Transactional Memory: The Big Idea

• Big idea I: **no locks, just shared data**

• 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];
void transfer(int id_from, int id_to, int 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];
void transfer(int id_from, int id_to, int 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)
    - Used to detect if anyone other thread (core) tries to write it
  - Locally buffer all of your writes (not yet visible to other processors)
  + **Local actions only: no lock acquire**

```c
struct acct_t { int bal; ... };
shared struct acct_t  accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
    begin_transaction();
    if (accts[id_from].bal >= amt) {
        accts[id_from].bal -= amt;
        accts[id_to].bal += amt;
    }
    end_transaction();
}
```
• **end_transaction**
  
  • Were any conflict detected? A conflict is:
    
    • Accesses from two **different** threads... to the **same** address...
      and at least one of them is a **write**
  
  • No? Commit transactions: commit writes
  
  • Yes? Abort transaction: restore checkpoint, try again

```c
struct acct_t { int bal; ... };
shared struct acct_t accts[MAX_ACCT];
void transfer(int id_from, int id_to, int amt) {
  begin_transaction();
  if (accts[id_from].bal >= amt) {
    accts[id_from].bal -= amt;
    accts[id_to].bal += amt;
  }
  end_transaction();
}
```
Transactional Memory Implementation

- How is read-set/write-set tracking implemented?
  - Track locations accessed using the cache
  - Leverage cache coherence protocol for conflict detection

- Read-set: additional “transactional read” bit per block
  - Set it on loads between begin_transaction and end_transaction
  - If other thread writes a block with bit set \(\rightarrow\) trigger abort
  - “Flash clear” all read bits 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)
  - If other thread reads or writes a block with bit set \(\rightarrow\) trigger abort
  - “Flash clear” all write bits to commit transaction
  - To abort transaction: invalidate all blocks with bit set
Transactional Execution

Thread 0

id_from = 241;
id_to = 37;

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

Thread 1

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 -= amt;
    acts[37].bal += amt;
}
end_transaction();
// no write to accts[241].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 -= amt;
    acts[118].bal += amt;
}
end_transaction();
// no write to accts[450].bal  // no write to accts[118].bal
// commit

• Critical sections execute in parallel
Summary

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- Memory consistency models
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
  - Best-effort hardware TM: Azul systems, Sun’s Rock processor
Speculative Lock Elision (SLE)

Alternatively, keep the locks, but...

... speculatively transactify lock-based programs in hardware

- **Speculative Lock Elision (SLE)** [Rajwar+, MICRO’01]
  - Captures most of the advantages of transactional memory...
    + No need to rewrite programs
    + Can always fall back on lock-based execution (overflow, I/O, etc.)

- Intel’s “Haswell” supports both SLE & best-effort TM

```c
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);
```
Roadmap Checkpoint

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- Memory consistency models
Shared Memory Example #1

- Initially: all variables zero (that is, x is 0, y is 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1  → y</td>
<td>store 1  → x</td>
</tr>
<tr>
<td>load x</td>
<td>load y</td>
</tr>
</tbody>
</table>

- What value pairs can be read by the two loads?
Initially: all variables zero (that is, x is 0, y is 0)

What value pairs can be read by the two loads?

What about (x=0, y=0)?  Nope... or can it?
Shared Memory Example #2

• Initially: all variables zero ("flag" is 0, "a" is 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 → a</td>
<td>loop: if (flag == 0) goto loop</td>
</tr>
<tr>
<td>store 1 → flag</td>
<td>load a</td>
</tr>
</tbody>
</table>

• What value can be read by "load a"?
**Shared Memory Example #2: “Answer”**

- **Initially:** all variables **zero** ("flag" is 0, "a" is 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 $\rightarrow$ a</td>
<td>loop: if (flag == 0) goto loop</td>
</tr>
<tr>
<td>store 1 $\rightarrow$ flag</td>
<td>load a</td>
</tr>
</tbody>
</table>

- What value can be read by “load a”?
  - “load a” can see the value “1”

- Can “load a” read the value zero?
  - Are you sure?
What is Going On?

- Reordering of memory operations to different addresses!
  - *Unlike* the sequential consistency model we’ve been assuming

- **In the compiler**
  - Compiler is generally allowed to re-order memory operations to different addresses
  - Many other compiler optimizations also cause problems

- **In the hardware**
  1. To tolerate write latency
     - Cores don’t wait for writes to complete (via store buffers)
     - And why should they? No reason to wait on non-threaded code
  2. To simplify out-of-order execution
Memory Consistency

- **Memory coherence**
  - Creates globally uniform (consistent) view...
  - Of **a single memory location** (in other words: cache blocks)
    - Not enough
      - Cache blocks 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
Recall: Write Misses and Store 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?

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

- **Store buffer vs. writeback-buffer**
  - Store buffer: “in front” of D$, for hiding store misses
  - Writeback buffer: “behind” D$, for hiding writebacks
Why? To Hide Store Miss Latency

- Why? Why Allow Such Odd Behavior?
  - Reason #1: hiding store miss latency

- Recall (back from caching unit)
  - Hiding store miss latency
  - How? Store buffer

- Said it would complicate multiprocessors
  - Yes. It does.
  - By allowing reordering of store and load (to different addresses)

- Example:
  - Both stores miss cache, are put in store buffer
  - Loads hit, receive value before store completes, sees “old” value

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 (\rightarrow y)</td>
<td>store 1 (\rightarrow x)</td>
</tr>
<tr>
<td>load (x)</td>
<td>load (y)</td>
</tr>
</tbody>
</table>
Shared Memory Example #1: Answer

- Initially: all variables zero (that is, x is 0, y is 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 → y</td>
<td>store 1 → x</td>
</tr>
<tr>
<td>load x</td>
<td>load y</td>
</tr>
</tbody>
</table>

- What value pairs can be read by the two loads?

- What about (x=0, y=0)? Yes! (for x86, SPARC, ARM, PowerPC)
Why? Simplify Out-of-Order Execution

• Why? Why Allow Such Odd Behavior?
  • Reason #2: simplifying out-of-order execution

• One key benefit of out-of-order execution:
  • Out-of-order execution of loads to (same or different) addresses

Two options for hardware designers:

• Option #1: **allow** this sort of “odd” reordering (“not my problem”)
• Option #2: hardware **detects & recovers** from such reorderings
  • Scan load queue (LQ) when cache block is invalidated

• Uh, oh.

• Uh, oh.

And store buffers on some systems reorder stores by same
  thread to different addresses (as in thread 1 above)
Simplifying Out-of-Order Execution

- Two options:
  - Option #1: allow this sort of “odd” reordering
  - Option #2: add more hardware, prevent these reorderings

- How to prevent?
  - Scan the Load Queue (LQ) on stores from other threads
  - Flush and rollback on conflict

- How to detect these stores from other threads?
  - Leverage cache coherence!
  - As long as the block remains in the cache...
    - Another core can’t write to it
  - Thus, anytime a block leaves the cache (invalidation or eviction)...
    - Scan the load queue. If any loads to the address have executed but not committed, squash the pipeline and restart
Shared Memory Example #2: Answer

- Initially: all variables zero (flag is 0, a is 0)

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>store 1 → a</td>
<td>loop: if (flag == 0) goto loop</td>
</tr>
<tr>
<td>store 1 → flag</td>
<td>load a</td>
</tr>
</tbody>
</table>

- What value can be read by “load a”?
  - “load a” can see the value “1”
- Can “load a” read the value zero? (same as last slide)
  - Yes! (for ARM, PowerPC, Itanium, and Alpha)
  - No! (for Intel/AMD x86, Sun SPARC, IBM 370)
    - Assuming the compiler didn’t reorder anything...
Restoring Order (Hardware)

- Sometimes we need ordering (mostly we don’t)
  - Prime example: ordering between “lock” and data

- How? insert **Fences (memory barriers)**
  - Special instructions, part of ISA

- Example
  - Ensure that loads/stores don’t cross synchronization operations
    ```
    lock acquire
    fence
    “critical section”
    fence
    lock release
    ```

- How do fences work?
  - They stall execution until write buffers are empty
  - Makes lock acquisition and release slow(er)

- **Use synchronization library, don’t write your own**
Restoring Order (Software)

- These slides have focused mostly on **hardware** reordering
  - But the compiler also reorders instructions (reason #3)
- **How do we tell the compiler to not reorder things?**
  - Depends on the language...

- **In Java:**
  - The built-in “synchronized” constructs informs the compiler to limit its optimization scope (prevent reorderings across synchronization)
  - Or, programmer uses “volatile” keyword to explicitly mark variables
  - Java compiler inserts the hardware-level ordering instructions

- **In C/C++:**
  - More murky, as pre-2011 language doesn’t define synchronization
  - Lots of hacks: “inline assembly”, volatile, atomic keyword (new!)
  - Programmer may need to explicitly insert hardware-level fences

- **Use synchronization library, don’t write your own**
Recap: Four Shared Memory Issues

1. **Cache coherence**
   - If cores have private (non-shared) caches
   - How to make writes to one cache “show up” in others?

2. **Parallel programming**
   - How does the programmer express the parallelism?

3. **Synchronization**
   - How to regulate access to shared data?
   - How to implement “locks”?

4. **Memory consistency models**
   - How to keep programmer sane while letting hardware optimize?
   - How to reconcile shared memory with compiler optimizations, store buffers, and out-of-order execution?
Summary

- Thread-level parallelism (TLP)
- Shared memory model
  - Multiplexed uniprocessor
  - Hardware multithreading
  - Multiprocessing
- Cache coherence
  - Valid/Invalid, MSI, MESI
- Parallel programming
- Synchronization
  - Lock implementation
  - Locking gotchas
  - Transactional memory
- Memory consistency models
EDITS FOR FUTURE

- Need less or more on “E” state
  - Really needs an example
Multiprocessing & Power Consumption

• Multiprocessing can be very power efficient

• 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 \textbf{1/6th the energy}
    • 5 cores * 1/30th = 1/6th

• Assumes parallel speedup (a difficult task)
  • Remember Ahmdal’s law
Sequential Consistency (SC)

A=flag=0;
Processor 0    Processor 1
A=1;           while (!flag); // spin
flag=1;        print A;

• 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

• Lamport’s definition: multiprocessor ordering...
  • Corresponds to some sequential interleaving of uniprocessor orders
  • Indistinguishable from multi-programmed uni-processor
Today’s Start-of-class Exercise

- Consider the following task
  - You have a list of bank account balances
    - <account_id, balance>
  - Some fast way to find balance given an account_id
  - You have a list of millions of “transfer” transactions to perform
    - <source_account_id, destination_account_id, amount>
- Brainstorm several ways you might perform this task...
  - alone
  - with a small group
  - with a group of 100
- What assumptions might simplify your task...
  - ... about the input data?
  - ... about the way the account information is stored?
Today’s Start-of-class Exercise

• Remember the assistant/researcher library analogy?

• New situation:
  • Collaboration among many researchers across the university
  • Researchers read & update various documents (books)
  • Goal: researcher always sees most recent version of a document

• How do all the assistants coordinate the sharing of books?
  • What are the issues/constraints/configurations to consider?
  • What would some reasonable goals be?
  • Give some possible algorithms

• Work in groups of three, will collect in a few minutes
Pre-Class Exercise

• 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:

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ld x</td>
<td>st 1 → y</td>
</tr>
<tr>
<td>ld y</td>
<td>st 1 → x</td>
</tr>
</tbody>
</table>

• What value pairs can be read by the two loads? (x, y) pairs:

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>st 1 → y</td>
<td>st 1 → x</td>
</tr>
<tr>
<td>ld x</td>
<td>ld y</td>
</tr>
</tbody>
</table>

• What value can be read by “Ld A” below?

<table>
<thead>
<tr>
<th>thread 1</th>
<th>thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>st 1 → A</td>
<td>while(flag == 0) { }</td>
</tr>
<tr>
<td>st 1 → flag</td>
<td>ld A</td>
</tr>
</tbody>
</table>
Test-and-Test-and-Set Lock Performance

Processor 0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0

// lock released by processor 2
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,(&lock)
A4: bnez r1,A0
CRITICAL_SECTION
A4: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0

Processor 1
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,(&lock)
A4: bnez r1,A0
CRITICAL_SECTION
A4: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0

- Processor 2 releases lock, informs (invalidates) processors 0 and 1
- Processors 0 and 1 race to acquire, processor 0 wins
Queue Lock Performance

Processor 0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
// lock released by processor 2
A0: ld r1,0(&lock)
A1: bnez r1,A0
A2: addi r1,1,r1
A3: t&s r1,(&lock)
A4: bnez r1,A0
CRITICAL_SECTION
A0: ld r1,0(&lock)
A1: bnez r1,A0

Processor 1
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0
A0: ld r1,0(&lock)
A1: bnez r1,A0

- Processor 2 releases lock, **informs only processor 0**
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
    • And multicore, of course

• 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
Revisiting Our Motivating Example

- 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`)

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)

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)

CPU0 CPU1 Mem

critical section (locks not shown)
### No-Cache, No-Problem

**Scenario I:** processors have no caches

- No problem

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: addi $r3,$r1,&amp;accts</td>
<td>0: addi $r3,$r1,&amp;accts</td>
</tr>
<tr>
<td>1: lw $r4,0($r3)</td>
<td>1: lw $r4,0($r3)</td>
</tr>
<tr>
<td>2: blt $r4,$r2,6</td>
<td>2: blt $r4,$r2,6</td>
</tr>
<tr>
<td>3: sub $r4,$r4,$r2</td>
<td>3: sub $r4,$r4,$r2</td>
</tr>
<tr>
<td>4: sw $r4,0($r3)</td>
<td>4: sw $r4,0($r3)</td>
</tr>
</tbody>
</table>

**Memory Accesses:**

- **CPU0:** Mem $500
- **CPU1:** Mem $500
- **Mem:** $400
- **Mem:** $300

Cache Incoherence

- Scenario II(a): processors have write-back caches
  - Potentially 3 copies of `accts[241].bal`: memory, two caches
  - Can get incoherent (inconsistent)
### Write-Through Doesn’t Fix It

**Scenario II(b): processors have write-through caches**

- This time only two (different) copies of `accts[241].bal`
- No problem? What if another withdrawal happens on processor 0?

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
<th>CPU0</th>
<th>CPU1</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: <code>addi $r3, $r1, &amp;accts</code></td>
<td>0: <code>addi $r3, $r1, &amp;accts</code></td>
<td>$400</td>
<td>$400</td>
<td>$400</td>
</tr>
<tr>
<td>1: <code>lw $r4, 0($r3)</code></td>
<td>1: <code>lw $r4, 0($r3)</code></td>
<td>$400</td>
<td>$400</td>
<td>$400</td>
</tr>
<tr>
<td>2: <code>blt $r4, $r2, 6</code></td>
<td>2: <code>blt $r4, $r2, 6</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3: <code>sub $r4, $r4, $r2</code></td>
<td>3: <code>sub $r4, $r4, $r2</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4: <code>sw $r4, 0($r3)</code></td>
<td>4: <code>sw $r4, 0($r3)</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU0: $500
CPU1: $500
Mem: $500
CPU0: $400
CPU1: $400
Mem: $400
CPU0: $400
CPU1: $300
Mem: $300
What To Do?

• No caches?
  – Too slow

• Make shared data uncachable?
  – Faster, but still too slow
  • Entire `accts` database is technically “shared”

• Flush all other caches on writes to shared data?
  • Can work well in some cases, but can make caches ineffective

• **Hardware cache coherence**
  • Rough goal: all caches have same data at all times
    + Minimal flushing, maximum caching → best performance
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

- **Each processors runs a state machine**

- Three processor-initiated events
  - **Ld**: load   **St**: store   **WB**: write-back

- Two remote-initiated events
  - **LdMiss**: read miss from *another* processor
  - **StMiss**: 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 & next slide)
    - 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 uses too much 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>Load Miss ➔ V</td>
<td>Store 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
  - **Memory sends data when no processor responds**
VI Protocol (Write-Back Cache)

- \texttt{lw} by processor 1 generates an “other load miss” event (LdMiss)
  - Processor 0 responds by sending its dirty copy, transitioning to \texttt{I}

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)

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)
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>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>Load Miss ➔ S</td>
<td>Store Miss ➔ M</td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Hit</td>
<td>Upgrade Miss ➔ M</td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Hit</td>
<td>Hit</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)

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: <code>addi $r3,$r1,&amp;accts</code></td>
<td></td>
</tr>
<tr>
<td>1: <code>lw $r4,0($r3)</code></td>
<td></td>
</tr>
<tr>
<td>2: <code>blt $r4,$r2,6</code></td>
<td></td>
</tr>
<tr>
<td>3: <code>sub $r4,$r4,$r2</code></td>
<td></td>
</tr>
<tr>
<td>4: <code>sw $r4,0($r3)</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>500</td>
</tr>
<tr>
<td>S:500</td>
<td></td>
<td>500</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M:400</th>
<th>500</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>S:400</th>
<th>S:400</th>
<th>400</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>I:</th>
<th>M:300</th>
<th>400</th>
</tr>
</thead>
</table>

- **`lw`** by processor 1 generates a “other load miss” event (LdMiss)
  - Processor 0 responds by sending its dirty copy, transitioning to S
- **`sw`** by processor 1 generates a “other store miss” event (StMiss)
  - Processor 0 responds by transitioning to I
Exclusive Clean Protocol Optimization

• 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

<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</td>
<td>Miss</td>
</tr>
<tr>
<td></td>
<td><strong>S or E</strong></td>
<td><strong>M</strong></td>
</tr>
<tr>
<td>Shared (S)</td>
<td>Hit</td>
<td>Upg Miss</td>
</tr>
<tr>
<td></td>
<td><strong>M</strong></td>
<td><strong>M</strong></td>
</tr>
<tr>
<td>Exclusive (E)</td>
<td>Hit</td>
<td>Hit</td>
</tr>
<tr>
<td></td>
<td><strong>M</strong></td>
<td><strong>S</strong></td>
</tr>
<tr>
<td>Modified (M)</td>
<td>Hit</td>
<td>Hit</td>
</tr>
<tr>
<td></td>
<td><strong>S</strong></td>
<td><strong>S</strong></td>
</tr>
</tbody>
</table>

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

- **Directories**: non-broadcast coherence protocol
  - Extend memory (or shared cache) to track caching information
  - For each physical cache block, 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 directory
    - Directory sends events only to processors **as needed**
  - Avoids non-scalable broadcast used by snooping protocols
  - For multicore with shared L3 cache, put directory info in cache tags

- For high-throughput, directory can be banked/partitioned
  - Use address to determine which bank/module holds a given block
  - That bank/module is called the “home” for the block
MSI Directory Protocol

- Processor side
  - Directory follows its own protocol

- Similar to bus-based MSI
  - Same three states
  - Same five actions (keep BR/BW names)
  - Minus red arcs/actions
    - Events that would not trigger action anyway
      + Directory won’t bother you unless you need to act

![Diagram showing MSI Directory Protocol states and transitions]

I: Instruction
M: Miss
S: Store

Transitions: LdMiss/StMiss, StMiss, WB, Load, Store, Load, LdMiss

Events: LdMiss, StMiss, WB, Load, Store
MSI Directory Protocol

- **ld** 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

• 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
Why? Simplify Out-of-Order Execution

- Two options for hardware designers:
  - Option #1: **allow** this sort of “odd” reordering (“not my problem”)
  - Option #2: hardware **detects & prevents** such reorderings

- How to prevent?
  - Scan the Load Queue (LQ) on stores from **other** threads
  - Flush and rollback on conflict

- How to detect these stores from other threads?
  - Leverage cache coherence!
  - As long as a block remains in a private per-core cache...
    - Another core can’t write to it!
  - Thus, anytime a block leaves the cache (invalidation or eviction)...
    - Scan the load queue. If any loads to the address have executed but not committed, squash the pipeline and restart
Coherence vs. Consistency

\[
\begin{align*}
A=0 & \quad \text{flag}=0 \\
\text{Processor 0} & \\
A=1; & \\
\text{flag}=1; & \\
\text{Processor 1} & \\
& \text{while (!flag);} \quad \text{// spin} \\
& \text{print A;} \\
\end{align*}
\]

- **Intuition says**: P1 prints A=1
- **Coherence says**: absolutely nothing
  - P1 can see P0’s write of `flag` before write of A!!! How?
    - P0 has a coalescing store buffer that reorders writes
    - Or out-of-order load execution
    - Or compiler reorders instructions
- Imagine trying to figure out why this code sometimes “works” and sometimes doesn’t
- **Real systems** are allowed to act in this strange manner
  - What is allowed? defined as part of the ISA and/or language
3 Classes of Memory Consistency Models

- **Sequential consistency (SC)** (MIPS, PA-RISC)
  - Formal definition of memory view programmers expect
    1. Processors see their own loads and stores in program order
    2. Processors see others’ loads and stores in program order
    3. All processors see same global load/store ordering
  - Corresponds to some sequential interleaving of uniprocessor orders
  - **Indistinguishable from multi-programmed uni-processor**

- **Processor consistency (PC)** (x86, SPARC)
  - Allows a in-order (FIFO) store buffer
    - Stores can be deferred, but must be put into the cache in order

- **Release consistency (RC)** (ARM, Itanium, PowerPC)
  - Allows an un-ordered coalescing store buffer
    - Stores can be put into cache in any order
  - Loads re-ordered, too.