Compsci 590.3: Introduction to Parallel Computing

Alvin R. Lebeck

Some slides from Computer Architecture slide deck
Mark Hill, Guri Sohi, Jim Smith, David Wood, Alvin Lebeck, Milo Martin, Dan Sorin, Krste Asanovic, Babak Falsafi, James Hoe, Mikko Lipasti, John Shen, T.N. Vijaykumar, and Ben Lee
Admin

• Homework
  ▪ Due 10/28
  ▪ Next (last) one is MPI

• Projects
  ▪ Project Work Days…

• Guest Lectures
  ▪ 11/2 Lars Nyland (NVIDIA)
  ▪ 11/9 Amanda Randles (BME)

• Outline
  • Shared memory review
  • Cache memory
  • Cache coherence
  • Memory consistency
Shared-Memory Multiprocessors

• Provide a shared-memory abstraction
  ▪ Familiar and efficient for programmers

P₁ P₂ P₃ P₄

Memory System
Shared-Memory Multiprocessors

- Provide a shared-memory abstraction
  - Familiar and efficient for programmers
Cache Memory

• Cache is a general concept in computer systems
  ▪ Web pages, search results, database queries, files, etc.

Caches in Computer Architecture
• Small fast memory close to processor
• May have multiple levels
• Exploits locality
  ▪ Reduces latency
  ▪ Absorbs bandwidth (reduces bandwith to lower levels)
Example: 1K Direct Mapped Cache

- **Cache Tag**: 0x0002fe
- **Cache Index**: 0x00
- **Byte Select**: 0x00

**Valid Bit**: 0

**Cache Tag**: 0xxxxxxx
- 0x000050
- 0x0000050
- 0x004440

**Cache Data**
- Byte 31: 
- Byte 63: 
- Byte 1023: 

**Cache Miss**
Xeon Phi Accelerator
Dual Socket Xeon Processor
Challenges in Shared Memory

• Synchronization
  § Atomic read/write operations

• Cache Coherence
  § “Common Sense”
  § P1-Read[X] → P1-Write[X] → P1-Read[X]  Read returns X
  § P1-Write[X] → P2-Read[X]  Read returns value written by P1
  § P1-Write[X] → P2-Write[X]  Writes serialized
  § All P’s see writes in same order

• Memory Consistency
  § What behavior should programmers expect from shared memory?
  § Provide a formal definition of memory behavior to programmer
  § Example: When will a written value be seen?
  § Example: P1-Write[X] … 10ps later… P2-Read[X]. What happens?
Example Program: ATM transaction

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

- Thread-level parallelism: thread per transaction
  - Good throughput
- My wife and I both withdraw at same time...
  - Two $100 withdrawals from account #241 at two ATM
An Example Parallel Execution

- 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`)
No-Cache, No-Problem

Scenario 1: processors have no caches
  - No problem
Cache Incoherence

**Scenario 2:** processors have write-back caches

- Potentially 3 copies of `accts[241].bal`: memory, p0$, p1$
- Can get incoherent (inconsistent)
Write-Thru Alone Doesn’t Help

Scenario 2: processors have write-thru caches
- This time only 2 (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>P0</th>
<th>P1</th>
<th>Mem</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>V:500</td>
<td>V:400</td>
<td>400</td>
</tr>
<tr>
<td>2: blt r4, r2, 6</td>
<td>2: blt r4, r2, 6</td>
<td>V:500</td>
<td>V:400</td>
<td>400</td>
</tr>
<tr>
<td>3: sub r4, r2, r4</td>
<td>3: sub r4, r2, r4</td>
<td>V:400</td>
<td>V:300</td>
<td>300</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 output_cash</td>
<td>5: call output_cash</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hardware Cache Coherence

- **Absolute coherence**
  - All copies have same data at all times
    - Hard to implement and slow
  - Not strictly necessary

- **Relative coherence**
  - Temporary incoherence OK (e.g., write-back)
  - As long as all loads get right values
    » i.e., no one looks at incoherent data

- **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
    » Finite State Machine per cache line
Cache Coherent Systems

• Provide Coherence Protocol
  ▪ States
  ▪ State transition diagram
  ▪ Actions

• Implement Coherence Protocol
  1. Determine when to invoke coherence protocol
  2. Find state of cache line to determine action
  3. Locate other cached copies
  4. Communicate with other cached copies (invalidate, update)

• Implementation Variants
  ▪ 1. is done in the same way for all systems. Maintain additional state per cache line. Invoke protocol based on state
  ▪ (2-4) have different approaches
Store Queue & Write Buffers

1. Store queues
   - Speculation

2. Write buffers
   - Hide store miss delays
   - Loads must search store buffer (in addition to D$)

3. Writeback buffers
   - Hides writeback delays
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

\[ A = \text{flag}=0; \]

Processor 0
A=1;
flag=1;

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

• Intuition says: P1 prints A=1
• Coherence says?
• Absolutely nothing!
  ▪ P1 can see P0’s write of \texttt{flag} before write of A!!! How?
    » Maybe coherence event of A is delayed somewhere in network
    » Maybe 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
Store Buffers & Consistency

\[
A = \text{flag} = 0; \\
\]

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

- Consider the following execution:
  - Processor 0’s write to A, misses the cache. Put in store buffer
  - Processor 0 keeps going
  - Processor 0 write “1” to flag hits, completes
  - Processor 1 reads flag… sees the value “1”
  - Processor 1 exits loop
  - Processor 1 prints “0” for A

- Ramification: store buffers can cause “strange” behavior
  - How strange depends on lots of things
Sequential Consistency (SC)

A=flag=0;

Processor 0
A=1;
flag=1;

Processor 1
while (!flag); // spin
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 definition: multiprocessor ordering…
  - Corresponds to some sequential interleaving of uniprocessor orders
  - I.e., indistinguishable from multi-programmed uni-processor
- Complex to enforce
Memory Consistency Models

- **Processor consistency (PC)** (x86, SPARC)
  - Allows a in-order store buffer (also Total Store Order TSO)
    » Stores can be deferred, but must be put into the cache in order

- **Release consistency (RC)** (ARM, Itanium, PowerPC)
  - Allows an un-ordered store buffer
    » Stores can be put into cache in any order
Restoring Order

• 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 lock acquire/release operation
    Acquire (lock)
    fence
    critical section
    fence
    Release (unlock)

• 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
Summary

- **Shared Memory**
  - All processors/threads have the same view of memory (shared global memory addresses)

- **Cache memory**
  - Small fast memory close to processor
  - Multiple levels

- **Cache coherence**
  - Technique (protocol) for ordering accesses to a single memory location to obtain a consistent view

- **Memory Consistency**
  - Specifies the allowable orderings of accesses for all memory locations…