Computer Systems: (Parallel)
Computer Architecture

Presenter: Sandeep K. S. Gupta

References:
- Chapter 5: Computer Architecture a Quantitative Approach, J. Hennessy and D. Patterson, Morgan Kauffman
- Chapter 5: Computer Organization and Design, D. Patterson and J. Hennessy, Morgan Kauffman,

Arizona State University
School of computing, informatics, and decision systems engineering
Agenda

- Thread Level-Parallelism
- Shared-Memory Multi-Processors
- Multi-Level Cache Issues (Recap)
- Cache Coherence Protocols
Thread-Level Parallelism

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled shared-memory multiprocessors

- For $n$ processors, need $n$ threads

- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit
Some Terminology

- **Core**: comprise a logical execution unit containing an L1 cache and functional units. Cores are able to independently execute programs or threads. Supercomputers have thousands of cores.

- **Chip**: refers to a physical integrated circuit (IC) on a computer. It is an execution unit with single- or multi-cores.

- **Socket**: is a physical connector on a computer motherboard that accepts a single physical chip. Many motherboards can have multiple sockets that can in turn accept multi-core chips.

- **Process**: is an independent program running on a computer. A process has a full stack of memory associated for its own use, and does not depend on another process for execution. MPI (Message Passing Interface) processes are true processes because they can run on independent machines or the same machine.

- **Thread**: is essentially a process that does not have a full stack of memory associated for it. The thread is tied to a parent process, and is merely an offshoot of execution. Typically thread processes must run on the same computer, but can execute simultaneously on separate cores of the same node. OpenMP parallelism uses threads for child processes.

- **Hyperthreading**: is a technology that preceded multi-core systems in which a single core would logically appear as multiple cores on the same chip. The "false cores" would gain some speed over a single core depending on the application. Most systems currently do not use hyperthreading technology, as this has been outdated by multi-core systems.

- **N-ways**: Multi-core compute nodes can be described by the number of execution units, or cores. A computer with 8 cores would be described as an 8-way node. This machine can have 8 independent processes running simultaneously. A 32-core system would be called a 32-way node.

- **Processor**: a processor could describe either a single execution core or a single physical multi-core chip. The context of use will define the meaning of the term.

Source: [http://kb.iu.edu/data/avfb.html](http://kb.iu.edu/data/avfb.html)
Shared-Memory Multiproc/core Archs.

- **Symmetric multiprocessors (SMP)**
  - Small number of cores
  - Share single memory with uniform memory latency

- **Distributed shared memory (DSM)**
  - Memory distributed among processors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks
Multilevel On-Chip Caches

Intel Nehalem 4-core processor

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache
Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size
The Memory Hierarchy (Recall)

- Common principles apply at all levels of the memory hierarchy
  - Based on notions of caching
- At each level in the hierarchy
  - Block placement
  - Finding a block
  - Replacement on a miss
  - Write policy
Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
  - Stall the CPU pipeline
  - Fetch block from next level of hierarchy
  - Instruction cache miss
    - Restart instruction fetch
  - Data cache miss
    - Complete data access
Write-Through

- On data-write hit, could just update the block in cache
  - But then cache and memory would be inconsistent
- Write through: also update memory
- But makes writes take longer
  - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - Effective CPI = 1 + 0.1\times100 = 11
- Solution: write buffer
  - Holds data waiting to be written to memory
  - CPU continues immediately
    - Only stalls on write if write buffer is already full
Write-Back

- Alternative: On data-write hit, just update the block in cache
  - Keep track of whether each block is dirty
- When a dirty block is replaced
  - Write it back to memory
  - Can use a write buffer to allow replacing block to be read first
Write Allocation

- What should happen on a write miss?
- Alternatives for write-through
  - Allocate on miss: fetch the block
  - Write around: don’t fetch the block
    - Since programs often write a whole block before reading it (e.g., initialization)
- For write-back
  - Usually fetch the block
Write Policy

- **Write-through**
  - Update both upper and lower levels
  - Simplifies replacement, but may require write buffer (for non-blocking writes).

- **Write-back**
  - Update upper level only
  - Update lower level when block is replaced
  - Need to keep more state information for each cache line

- **Virtual memory**
  - Only write-back is feasible, given disk write latency
Cache Control

- Example cache characteristics
  - Direct-mapped, write-back, write allocate
  - Block size: 4 words (16 bytes)
  - Cache size: 16 KB (1024 blocks)
  - 32-bit byte addresses
  - Valid bit and dirty bit per block
  - Blocking cache
    - CPU waits until access is complete

<table>
<thead>
<tr>
<th>31</th>
<th>10</th>
<th>9</th>
<th>4</th>
<th>3</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
<td>Offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 bits</td>
<td>10 bits</td>
<td>4 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interface Signals

CPU

| Read/Write | Valid | Address (32) | Write Data (32) | Read Data (32) | Ready |

Cache

| Read/Write | Valid | Address (32) | Write Data (128) | Read Data (128) | Ready |

Memory

Multiple cycles per access
Direct-Mapped Cache Organization
Set Associative Cache Organization
Cache Controller FSM

Could partition into separate states to reduce clock cycle time.
**Cache Coherence Problem**

- Suppose two CPU cores share a physical address space
  - Write-through caches
- Processors may see different values through their caches:

<table>
<thead>
<tr>
<th>Time step</th>
<th>Event</th>
<th>CPU A's cache</th>
<th>CPU B's cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CPU A writes 1 to X</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Coherence Defined

- Informally: Reads return most recently written value
- Formally:
  - P writes X; P reads X (no intervening writes)
    \[ \Rightarrow \text{read returns written value} \]
  - \( P_1 \) writes X; \( P_2 \) reads X (sufficiently later)
    \[ \Rightarrow \text{read returns written value} \]
      - c.f. CPU B reading X after step 3 in example
  - \( P_1 \) writes X, \( P_2 \) writes X
    \[ \Rightarrow \text{all processors see writes in the same order} \]
      - End up with the same final value for X
Cache Coherence Protocols

- Operations performed by caches in multiprocessors to ensure coherence
  - Migration of data to local caches
    - Reduces bandwidth for shared memory
  - Replication of read-shared data
    - Reduces contention for access
- Snooping protocols
  - Each cache monitors bus reads/writes
- Directory-based protocols
  - Caches and memory record sharing status of blocks in a directory
Memory Consistency

- When are writes seen by other processors
  - “Seen” means a read returns the written value
  - Can’t be instantaneously

- Assumptions
  - A write completes only when all processors have seen it
  - A processor does not reorder writes with other accesses

- Consequence
  - P writes X then writes Y
    \[ \Rightarrow \text{all processors that see new Y also see new X} \]
  - Processors can reorder reads, but not writes
Summary Cache Coherence

- **Coherence**
  - All reads by any processor must return the most recently written value
  - Writes to the same location by any two processors are seen in the same order by all processors

- **Consistency**
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A
Snoopy Coherence Protocols

- **Write invalidate**
  - On write, invalidate all other copies
  - Use bus itself to serialize
    - Write cannot complete until bus access is obtained

<table>
<thead>
<tr>
<th>Processor activity</th>
<th>Bus activity</th>
<th>Contents of processor A’s cache</th>
<th>Contents of processor B’s cache</th>
<th>Contents of memory location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Processor B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Processor A writes 1</td>
<td>Invalidation for X</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>to X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor B reads X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Write update**
  - On write, update all copies
Snoopy Coherence Protocols

- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor

- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive
<table>
<thead>
<tr>
<th>Request</th>
<th>Source</th>
<th>State of addressed cache block</th>
<th>Type of cache action</th>
<th>Function and explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read hit</td>
<td>Processor</td>
<td>Shared or modified</td>
<td>Normal hit</td>
<td>Read data in local cache.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Processor</td>
<td>Invalid</td>
<td>Normal miss</td>
<td>Place read miss on bus.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Replacement</td>
<td>Address conflict miss: place read miss on bus.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Processor</td>
<td>Modified</td>
<td>Replacement</td>
<td>Address conflict miss: write-back block, then place read miss on bus.</td>
</tr>
<tr>
<td>Write hit</td>
<td>Processor</td>
<td>Modified</td>
<td>Normal hit</td>
<td>Write data in local cache.</td>
</tr>
<tr>
<td>Write hit</td>
<td>Processor</td>
<td>Shared</td>
<td>Coherence</td>
<td>Place invalidate on bus. These operations are often called upgrade or ownership misses, since they do not fetch the data but only change the state.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Invalid</td>
<td>Normal miss</td>
<td>Place write miss on bus.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Shared</td>
<td>Replacement</td>
<td>Address conflict miss: place write miss on bus.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Processor</td>
<td>Modified</td>
<td>Replacement</td>
<td>Address conflict miss: write-back block, then place write miss on bus.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Bus</td>
<td>Shared</td>
<td>No action</td>
<td>Allow shared cache or memory to service read miss.</td>
</tr>
<tr>
<td>Read miss</td>
<td>Bus</td>
<td>Modified</td>
<td>Coherence</td>
<td>Attempt to share data: place cache block on bus and change state to shared.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>Bus</td>
<td>Shared</td>
<td>Coherence</td>
<td>Attempt to write shared block; invalidate the block.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Bus</td>
<td>Shared</td>
<td>Coherence</td>
<td>Attempt to write shared block; invalidate the cache block.</td>
</tr>
<tr>
<td>Write miss</td>
<td>Bus</td>
<td>Modified</td>
<td>Coherence</td>
<td>Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.</td>
</tr>
</tbody>
</table>
Snoopy Coherence Protocols

- Invalid
  - CPU read
    - Place read miss on bus
  - CPU write
    - Place write miss on bus

- Shared (read only)
  - CPU read hit
  - CPU read miss
  - CPU write miss
  - Place write miss on bus

- Exclusive (read/write)
  - CPU write miss
  - Write-back cache block
  - Place write miss on bus

- Cache state transitions based on requests from CPU

- Invalid
  - Write miss for this block
  - Write-back block:
    - abort memory access
    - Write-back block:
      - abort memory access

- Exclusive (read/write)
  - Read miss for this block
  - Cache state transitions based on requests from the bus