MULTIPROCESSORS AND THREAD-LEVEL PARALLELISM

B649
Parallel Architectures and Programming
Motivation behind Multiprocessors

• Limitations of ILP (as already discussed)
• Growing interest in servers and server-performance
• Growth in data-intensive applications
• Increasing desktop performance relatively unimportant
• Effectiveness of multiprocessors for server applications
• Leveraging design investment by replication
Flynn’s Classification of Parallel Architectures

- **SISD**: Single Instruction Single Data stream
  - uniprocessors
- **SIMD**: Single Instruction Multiple Data streams
  - suitable for data parallelism
  - Intel’s multimedia extensions, vector processors
  - growing popularity in graphics applications
- **MISD**: Multiple instruction Single Data stream
  - no commercial multiprocessor to date
- **MIMD**: Multiple Instruction Multiple Data streams
  - suitable for thread-level parallelism
MIMD

- Architecture of choice for general-purpose multiprocessors
- Offers flexibility
- Can leverage the design investment in uniprocessors
- Can use off-the-shelf processors
  - “COTS” (Commercial, Off-The-Shelf) processors
- Examples
  - Clusters
    - commodity and custom clusters
  - Multicores
Shared-Memory Multiprocessors

- Processor
  - One or more levels of cache
- Main memory

- Processor
  - One or more levels of cache

- Processor
  - One or more levels of cache

- Processor
  - One or more levels of cache

- I/O system

© 2007 Elsevier, Inc. All rights reserved.
Distributed-Memory Multiprocessors
Models for Memory and Communication

• Memory architecture
  ★ shared memory
    ★ Uniform Memory Access (UMA)
    ★ Symmetric (Shared-Memory) Multiprocessors (SMPs)
  ★ distributed memory
    ★ Non-Uniform Memory Access (NUMA)

• Communication architecture (programming)
  ★ shared memory
  ★ message-passing
Other Ways to Categorize Parallel Programming

- data vs task parallel
- client-server vs p-to-p/master-slave vs symm.
- threads vs producer-consumer
- tightly vs loosely coupled
- shared mem vs msg passing
- recursive vs iterative
- course vs fine grained
- SPMD vs MPMD

Parallel Programs
<table>
<thead>
<tr>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>cache</td>
</tr>
<tr>
<td>virtual memory</td>
</tr>
<tr>
<td>memory stall cycles</td>
</tr>
<tr>
<td>direct mapped</td>
</tr>
<tr>
<td>valid bit</td>
</tr>
<tr>
<td>block address</td>
</tr>
<tr>
<td>write through</td>
</tr>
<tr>
<td>instruction cache</td>
</tr>
<tr>
<td>average memory access time</td>
</tr>
<tr>
<td>cache hit</td>
</tr>
<tr>
<td>page</td>
</tr>
<tr>
<td>miss penalty</td>
</tr>
<tr>
<td>fully associative</td>
</tr>
<tr>
<td>dirty bit</td>
</tr>
<tr>
<td>block offset</td>
</tr>
<tr>
<td>write back</td>
</tr>
<tr>
<td>data cache</td>
</tr>
<tr>
<td>hit time</td>
</tr>
<tr>
<td>cache miss</td>
</tr>
<tr>
<td>page fault</td>
</tr>
<tr>
<td>miss rate</td>
</tr>
<tr>
<td>n-way set associative</td>
</tr>
<tr>
<td>least-recently used</td>
</tr>
<tr>
<td>tag field</td>
</tr>
<tr>
<td>write allocate</td>
</tr>
<tr>
<td>unified cache</td>
</tr>
<tr>
<td>misses per instruction</td>
</tr>
<tr>
<td>block</td>
</tr>
<tr>
<td>locality</td>
</tr>
<tr>
<td>address trace</td>
</tr>
<tr>
<td>set</td>
</tr>
<tr>
<td>random replacement</td>
</tr>
<tr>
<td>index field</td>
</tr>
<tr>
<td>no-write allocate</td>
</tr>
<tr>
<td>write buffer</td>
</tr>
<tr>
<td>write stall</td>
</tr>
</tbody>
</table>
# Memory Hierarchy

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>registers</td>
<td>cache</td>
<td>main memory</td>
<td>disk storage</td>
</tr>
<tr>
<td>Typical size</td>
<td>&lt; 1 KB</td>
<td>&lt; 16 MB</td>
<td>&lt; 512 GB</td>
<td>&gt; 1 TB</td>
</tr>
<tr>
<td>Implementation technology</td>
<td>custom memory with multiple ports, CMOS</td>
<td>on-chip or off-chip CMOS SRAM</td>
<td>CMOS DRAM</td>
<td>magnetic disk</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.25–0.5</td>
<td>0.5–25</td>
<td>50–250</td>
<td>5,000,000</td>
</tr>
<tr>
<td>Bandwidth (MB/sec)</td>
<td>50,000–500,000</td>
<td>5000–20,000</td>
<td>2500–10,000</td>
<td>50–500</td>
</tr>
<tr>
<td>Managed by</td>
<td>compiler</td>
<td>hardware</td>
<td>operating system</td>
<td>operating system/operator</td>
</tr>
<tr>
<td>Backed by</td>
<td>cache</td>
<td>main memory</td>
<td>disk</td>
<td>CD or tape</td>
</tr>
</tbody>
</table>
Four Memory-Hierarchy Questions

- Where can a block be placed in the upper level? ★ block placement
- How is a block found if it is in the upper level? ★ block identification
- Which block should be replaced on a miss? ★ block replacement
- What happens on a write? ★ write strategy
Where Can a Block Be Placed in a Cache?

- Only one place for each block
  - direct mapped
    \[(\text{Block address}) \ MOD \ (\text{Number of blocks in cache})\]

- Anywhere in the cache
  - fully associative

- Restricted set of places
  - set associative
    \[(\text{Block address}) \ MOD \ (\text{Number of sets in cache})\]
Example

- Fully associative: block 12 can go anywhere
- Direct mapped: block 12 can go only into block 4 (12 mod 8)
- Set associative: block 12 can go anywhere in set 0 (12 mod 4)
How is a Block Found if it is in Cache?

- “Tags” in each cache block gives the block address
  - all possible tags searched in parallel (associative memory)
  - *valid bit* tells whether a tag match valid

Fields in a memory address

<table>
<thead>
<tr>
<th>Block address</th>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
</table>

- No “index” field in fully associative caches
Which Block Should be Replaced on a Miss?

• Random
  ★ easy to implement

• Least-recently used (LRU)
  ★ idea: rely on the past to predict the future
  ★ replace the block unused for the longest time

• First in, First out (FIFO)
  ★ approximates LRU (oldest, rather than least recently used)
  ★ simpler to implement
## Comparison of Replacement Policies

Data cache misses per 1000 instructions on five SPECint2000 and five SPECfp2000 benchmarks

<table>
<thead>
<tr>
<th>Size</th>
<th>Two-way</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Random</td>
<td>FIFO</td>
</tr>
<tr>
<td>16 KB</td>
<td>114.1</td>
<td>117.3</td>
<td>115.5</td>
</tr>
<tr>
<td>64 KB</td>
<td>103.4</td>
<td>104.3</td>
<td>103.9</td>
</tr>
<tr>
<td>256 KB</td>
<td>92.2</td>
<td>92.1</td>
<td>92.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Four-way</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Random</td>
<td>FIFO</td>
</tr>
<tr>
<td></td>
<td>111.7</td>
<td>115.1</td>
<td>113.3</td>
</tr>
<tr>
<td></td>
<td>102.4</td>
<td>102.3</td>
<td>103.1</td>
</tr>
<tr>
<td></td>
<td>92.1</td>
<td>92.1</td>
<td>92.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Eight-way</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LRU</td>
<td>Random</td>
<td>FIFO</td>
</tr>
<tr>
<td></td>
<td>109.0</td>
<td>111.8</td>
<td>110.4</td>
</tr>
<tr>
<td></td>
<td>99.7</td>
<td>100.5</td>
<td>100.3</td>
</tr>
<tr>
<td></td>
<td>92.1</td>
<td>92.1</td>
<td>92.5</td>
</tr>
</tbody>
</table>
What Happens on a Write?

• Reads dominate
  ★ 7% of the overall memory traffic are writes
  ★ 28% of the data cache traffic are writes

• Write takes longer
  ★ reading cache line and validity check can be parallel
  ★ reads can read the whole line, write must modify only the specified bytes
Handling Writes

• Write strategy
  ★ write through
    ★ write to cache block and to the block in the lower-level memory
  ★ write back
    ★ write only to cache block, update the lower-level memory when block replaced

• Block allocation strategy
  ★ write allocate
    ★ allocate a block on cache miss
  ★ no-write allocate
    ★ do not allocate, no affect on cache
Example: Opteron Data Cache

Diagram showing the Opteron data cache architecture with blocks, tags, indexes, and data transfer paths.
Terminology

cache
virtual memory
memory stall cycles
direct mapped
valid bit
block address
write through
instruction cache
average memory access time
cache hit
page
miss penalty

fully associative
dirty bit
block offset
write back
data cache
hit time
cache miss
page fault
miss rate
n-way set associative
least-recently used
tag field

write allocate
unified cache
misses per instruction
block
locality
address trace
set
random replacement
index field
no-write allocate
write buffer
write stall
SYMMETRIC SHARED-MEMORY: CACHE COHERENCE
We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini, describing Intel’s future direction at the Intel Developers Forum in 2005
Quotes

The turning away from conventional organization came in the middle 1960s, when the law of diminishing returns began to take effect in the effort to increase the operational speed of a computer ... Electronic circuits are ultimately limited in their speed of operation by the speed of light ... and many of the circuits were already operating in the nanosecond range.

W. Jack Bouknight et al.
The Illiac IV System (1972)

We are dedicating all of our future product development to multicore designs. We believe this is a key inflection point for the industry.

Intel President Paul Otellini,
describing Intel’s future direction at the Intel Developers Forum in 2005
Multiprocessor Cache Coherence

X is not in any cache initially. The caches are write-through.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache contents for CPU A</th>
<th>Cache contents for CPU B</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A stores 0 into X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Multiprocessor Cache Coherence

X is not in any cache initially. The caches are write-through.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache contents for CPU A</th>
<th>Cache contents for CPU B</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A stores 0 into X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Proposed definition: Memory system is coherent if any read of a data item returns the most recently written value of that data item.
Multiprocessor Cache Coherence

X is not in any cache initially. The caches are write-through.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>Cache contents for CPU A</th>
<th>Cache contents for CPU B</th>
<th>Memory contents for location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>CPU A reads X</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>CPU B reads X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>CPU A stores 0 into X</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Proposed definition: Memory system is coherent if any read of a data item returns the most recently written value of that data item.

Too simplistic!
Coherency: Take 2

• A memory system is coherent if:
Coherency: Take 2

• A memory system is coherent if:

1. Writes and Reads by one processor

\[ P \xrightarrow{x} M \]

\[ P \xleftarrow{x} M \]
Coherency: Take 2

• A memory system is coherent if:

1. Writes and Reads by one processor

2. Writes and Reads by two processors
Coherency: Take 2

- A memory system is coherent if:

1. Writes and Reads by one processor

   ![Diagram 1](image1)

2. Writes and Reads by two processors

   ![Diagram 2](image2)

3. Writes by two processors (serialization)

   ![Diagram 3](image3)

\[x\]
What Needs to Happen for Coherence?

• Migration
  ★ data can move to local cache, when needed

• Replication
  ★ data may be replicated in local cache, when needed

• Need a protocol to maintain the coherence property
  ★ specialized hardware
Coherence Protocols

• Directory based
  ★ central location (directory) maintains the sharing state of a block of physical memory
  ★ slightly higher implementation cost
  ★ scalable
  ★ most used for distributed-memory multiprocessors

• Snooping
  ★ each cache maintains sharing state of the blocks it contains
  ★ shared broadcast medium (e.g., bus)
  ★ each cache snoops on the medium to determine whether they a copy of the block that is requested
  ★ most used for shared-memory multiprocessors
Snooping Protocols: Handling Writes

- **Write invalidate**
  - write requires exclusive access
  - any copy held by the reading processor is **invalidated**
  - if two processors attempt to write, one wins the race

- **Write update** (or write broadcast)
  - **update** all the cached copies of a data item when written
  - consumes substantially more bandwidth than invalidating-based protocol
  - not used in recent multiprocessors
# Example of Invalidation Protocol

## Write-back caches

<table>
<thead>
<tr>
<th>Processor activity</th>
<th>Bus activity</th>
<th>Contents of CPU A’s cache</th>
<th>Contents of CPU B’s cache</th>
<th>Contents of memory location X</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU A reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU A writes a 1 to X</td>
<td>Invalidation for X</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU B reads X</td>
<td>Cache miss for X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Write Invalidate Protocol: Observations

• Serialization through access to the broadcast medium

• Need to locate data item upon miss
  ★ simple on write-through caches
    ★ write-buffers may complicate this
    ★ write-through increases the memory bandwidth requirement
  ★ more complex on write-back caches
    ★ caches snoop for read addresses, supply matching dirty block
    ★ no need to write back dirty block if cached elsewhere
    ★ preferred approach on most modern multiprocessors, due to lower memory bandwidth requirements

• Cache tags and valid bits can do double duty

• Additional bit to indicate whether block shared

• Desirable to reduce contention on cache between processor and snooping
Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write
Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

• Idea: Shared read, exclusive write
Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

- Idea: Shared read, exclusive write

- Read
  - Hit: get from local cache
  - Miss: get from memory or another processor’s cache; write-back existing block if needed
Invalidation-Based Coherence Protocol for Write-Back Caches with Allocate on Write

• Idea: Shared read, exclusive write

• Read
  ★ Hit: get from local cache
  ★ Miss: get from memory or another processor’s cache; write-back existing block if needed

• Write
  ★ Hit: write in cache, mark the block exclusive (invalidate copies at other processors)
  ★ Miss: get from memory or another processor’s cache; write-back existing block if needed
**Cache Coherence Mechanism (MESI or MOESI)**

<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 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 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 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 block that is shared; 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.</td>
</tr>
</tbody>
</table>
Write Invalidate Cache Coherence Protocol for Write-Back Caches
Write Invalidate Cache Coherence Protocol for Write-Back Caches
Interconnection Network, Instead of Bus
SYMMETRIC SHARED-MEMORY: PERFORMANCE
Performance Issues

• Cache misses
  ★ capacity
  ★ compulsory
  ★ conflict
• Cache coherence
  ★ true-sharing misses
  ★ false-sharing
Performance Issues

• Cache misses
  ★ capacity
  ★ compulsory
  ★ conflict

• Cache coherence
  ★ true-sharing misses
  ★ false-sharing

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Write x1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Read x2</td>
</tr>
<tr>
<td>3</td>
<td>Write x1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Write x2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Read x2</td>
</tr>
</tbody>
</table>
Performance: Alpha-Server

• Machine
  ★ Alpha-Server 4100, 4 processors: processor Alpha 21164 (four issue)

• Three-level cache
  ★ L1: 8KB direct-mapped, separate instruction and data, 32-byte block size, write through, on-chip
  ★ L2: 96KB 3-way set-associative, unified, 32-byte block size, write back, on-chip
  ★ L3: 2MB direct-mapped, unified, 64-byte block size, write back, off-chip

• Latencies
  ★ L2: 7 cycles, L3: 21 cycles, memory: 80 cycles
Performance: Commercial Workload

• Online Transaction-Processing (OLTP)
  ★ modeled after TPC-B
  ★ client-server

• Decision Support System (DSS)
  ★ modeled after TPC-D
  ★ long-running queries against large complex data structures (obsolete)

• Web index search
  ★ AltaVista, using 200 GB memory-mapped database

• I/O time ignored (substantial for these applications)
Execution Time Breakdown

- **OLTP**
  - Instruction execution: 20%
  - L2 access: 10%
  - L3 access: 5%
  - Memory access: 15%
  - Other stalls: 5%

- **DSS**
  - Instruction execution: 40%
  - L2 access: 20%
  - L3 access: 15%
  - Memory access: 10%
  - Other stalls: 5%

- **AV**
  - Instruction execution: 50%
  - L2 access: 30%
  - L3 access: 15%
  - Memory access: 5%
  - Other stalls: 0%

© 2007 Elsevier, Inc. All rights reserved.
OLTP with Different Cache Sizes

Diagram showing normalized execution time for different L3 cache sizes (1 MB, 2 MB, 4 MB, 8 MB). The graph indicates the percentage of time spent on various activities such as instruction execution, L2/L3 cache access, memory access, PAL code, and idle state. The x-axis represents the L3 cache size in MB, and the y-axis represents the normalized execution time.
OLTP: Contributing Causes of L3 Cycles
OLTP Mem. Access Cycles with Processor Count
OLPT Misses with L3 Cache Block Size
Multiprogramming and OS Workload

• Models user and OS activities

• Andrew benchmark, emulates software development
  ★ compiling
  ★ installing object files in a library
  ★ removing object files

• Memory hierarchy
  ★ L1 instruction cache: 32KB, 2-way set-associative, 64-byte block; L1 data cache: 32KB 2-way set-associative, 32-byte block
  ★ L2: 1MB unified, 2-way set assoc., 128-byte block
  ★ Memory: 100 clock cycles access
## Distribution of Execution Time

<table>
<thead>
<tr>
<th></th>
<th>User execution</th>
<th>Kernel execution</th>
<th>Synchronization wait</th>
<th>CPU idle (waiting for I/O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>% instructions executed</td>
<td>27</td>
<td>3</td>
<td>1</td>
<td>69</td>
</tr>
<tr>
<td>% execution time</td>
<td>27</td>
<td>7</td>
<td>2</td>
<td>64</td>
</tr>
</tbody>
</table>
L1 Size and L1 Block Size

- Kernel miss rate decreases as cache size increases.
- User miss rate decreases as cache size increases.

- Kernel miss rate decreases as block size increases.
- User miss rate decreases as block size increases.
Kernel Data Cache Miss Rates

The bar charts illustrate the miss rates for different cache sizes and block sizes. The bars are divided into three categories: compulsory, coherence, and capacity misses.

- **Cache size (KB):**
  - 32 KB: Miss rates range from 0% to 7%.
  - 64 KB: Miss rates range from 1% to 6%.
  - 128 KB: Miss rates range from 2% to 5%.
  - 256 KB: Miss rates range from 3% to 4%.

- **Block size (bytes):**
  - 16 bytes: Miss rates range from 0% to 9%.
  - 32 bytes: Miss rates range from 1% to 8%.
  - 64 bytes: Miss rates range from 2% to 7%.
  - 128 bytes: Miss rates range from 3% to 6%.

© 2007 Elsevier, Inc. All rights reserved.
Memory Traffic Per Data Reference

![Graph showing memory traffic per data reference against block size (bytes). The graph compares kernel traffic and user traffic. Kernel traffic increases significantly with block size, while user traffic remains relatively stable.](image)
DISTRIBUTED SHARED-MEMORY: DIRECTORY-BASED COHERENCE
Distributed Memory
Distributed Memory + Directories
Directory-Based Protocol States

- **Shared**
  - one or more processors have the block cached
  - memory has up to date value
- **Uncached**
  - no processor has a copy of the cache block
- **Modified**
  - exactly one processor has a copy of the cache block
  - memory copy is out of date
  - the processor with the copy is the block’s **owner**
### Possible Messages

<table>
<thead>
<tr>
<th>Message type</th>
<th>Source</th>
<th>Destination</th>
<th>Message contents</th>
<th>Function of this message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read miss</td>
<td>local cache</td>
<td>home directory</td>
<td>P, A</td>
<td>Processor P has a read miss at address A; request data and make P a read sharer.</td>
</tr>
<tr>
<td>Write miss</td>
<td>local cache</td>
<td>home directory</td>
<td>P, A</td>
<td>Processor P has a write miss at address A; request data and make P the exclusive owner.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>local cache</td>
<td>home directory</td>
<td>A</td>
<td>Request to send invalidates to all remote caches that are caching the block at address A.</td>
</tr>
<tr>
<td>Invalidate</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Invalidate a shared copy of data at address A.</td>
</tr>
<tr>
<td>Fetch</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.</td>
</tr>
<tr>
<td>Fetch/invalidate</td>
<td>home directory</td>
<td>remote cache</td>
<td>A</td>
<td>Fetch the block at address A and send it to its home directory; invalidate the block in the cache.</td>
</tr>
<tr>
<td>Data value reply</td>
<td>home directory</td>
<td>local cache</td>
<td>D</td>
<td>Return a data value from the home memory.</td>
</tr>
<tr>
<td>Data write back</td>
<td>remote cache</td>
<td>home directory</td>
<td>A, D</td>
<td>Write back a data value for address A.</td>
</tr>
</tbody>
</table>
States for Individual Caches

- **Invalid**: CPU read
  - Send read miss message
- **Shared (read only)**
  - CPU read hit
- **Modified (read/write)**
  - CPU write
  - CPU write miss
  - CPU read hit
  - Fetch invalidate
  - Send write miss message
- **Data write back**: CPU read miss
  - Data write back; read miss
  - Fetch
  - Send invalidate message
- **Write miss**: Data write back
  - CPU write hit
  - Send write miss message

© 2007 Elsevier Inc. All rights reserved.
States for Directories

- **Uncached**
  - Data value reply; *Sharers* = \{P\}
  - Read miss
  - Write miss

- **Exclusive (read/write)**
  - Data write back
  - Data value reply; *Sharers* = \{P\}
  - Read miss
  - Fetch; data value reply; *Sharers* = *Sharers* + \{P\}
  - Invalidate; *Sharers* = *Sharers*
  - Fetch/Invalidate
  - Data value reply
    - *Sharers* = \{P\}

- **Shared (read only)**
  - Read miss
  - Write miss

- **States Transition Diagram**
  - From Uncached to Shared
  - From Uncached to Exclusive
  - From Exclusive to Uncached
  - From Exclusive to Shared

- **Shared**
  - *Sharers* = \{P\}
  - Data value reply
    - *Sharers* = *Sharers* + \{P\}

- **Exclusive**
  - *Sharers* = \{P\}
  - Data value reply
    - *Sharers* = *Sharers*
  - Write miss

- **Uncached**
  - Data value reply; *Sharers* = \{P\}
  - Read miss
  - Write miss
SYNCHRONIZATION
Basic Idea

• Hardware support for atomically reading and writing a memory location
  ★ enables software to implement locks
  ★ a variety of synchronizations possible with locks
• Multiple (equivalent) approaches possible
• Synchronization libraries on top of hardware primitives
Some Examples

• Atomic exchange
  ★ exchange a register and memory value atomically
  ★ return the register value if failed, memory value if succeeded

• Test-and-set
  ★ test a memory location and set its value if the test passed

• Fetch-and-increment
Paired Instructions

• Problems with single atomic operations
  ★ complicates coherence

• Alternative: pair of special load and store instructions
  ★ load linked or load locked
  ★ store conditional
  ★ can implement atomic exchange with the pair

<table>
<thead>
<tr>
<th>try:</th>
<th>MOV</th>
<th>R3,R4</th>
</tr>
</thead>
</table>
|      | LL   | R2,0(R1)| load linked
|      | SC   | R3,0(R1)| store conditional
| BEQZ | R3,try | branch store fails
| MOV  | R4,R2 | put load value in R4 |
Other Primitives can also be built

• Atomic exchange

<table>
<thead>
<tr>
<th>try:</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>try:</td>
<td>MOV R3,R4</td>
<td>mov exchange value</td>
</tr>
<tr>
<td>LL</td>
<td>R2,0(R1)</td>
<td>load linked</td>
</tr>
<tr>
<td>SC</td>
<td>R3,0(R1)</td>
<td>store conditional</td>
</tr>
<tr>
<td>BEQZ</td>
<td>R3,try</td>
<td>branch store fails</td>
</tr>
<tr>
<td>MOV</td>
<td>R4,R2</td>
<td>put load value in R4</td>
</tr>
</tbody>
</table>

• Fetch-and-increment

<table>
<thead>
<tr>
<th>try:</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>try:</td>
<td>LL R2,0(R1)</td>
<td>load linked</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R3,R2,#1</td>
<td>increment</td>
</tr>
<tr>
<td>SC</td>
<td>R3,0(R1)</td>
<td>store conditional</td>
</tr>
<tr>
<td>BEQZ</td>
<td>R3,try</td>
<td>branch store fails</td>
</tr>
</tbody>
</table>

• Implemented with a link register to track the address of LL instruction
Implementing Spin Locks: Uncached

lockit:

- DADDUI R2, R0, #1
- EXCH R2, 0(R1)
- BNEZ R2, lockit

; atomic exchange
; already locked?
Implementing Spin Locks: Cached

| lockit:    | LD   | R2,0(R1) | ;load of lock    |
|           | BNEZ | R2,lockit | ;not available-spin |
|           | DADDUI | R2,R0,#1 | ;load locked value |
|           | EXCH | R2,0(R1) | ;swap            |
|           | BNEZ | R2,lockit | ;branch if lock wasn't 0 |
### Cache Coherence Steps

<table>
<thead>
<tr>
<th>Step</th>
<th>Processor P0</th>
<th>Processor P1</th>
<th>Processor P2</th>
<th>Coherence state of lock</th>
<th>Bus/directory activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Has lock</td>
<td>Spins, testing if lock = 0</td>
<td>Spins, testing if lock = 0</td>
<td>Shared</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Set lock to 0</td>
<td>(Invalidate received)</td>
<td>(Invalidate received)</td>
<td>Exclusive (P0)</td>
<td>Write invalidate of lock variable from P0</td>
</tr>
<tr>
<td>3</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td>Cache miss</td>
<td>Shared</td>
<td>Bus/directory services P2 cache miss; write back from P0</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Lock = 0</td>
<td>Lock = 0</td>
<td>Shared</td>
<td>Cache miss for P2 satisfied</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>Lock = 0</td>
<td>Executes swap, gets cache miss</td>
<td>Shared</td>
<td>Cache miss for P1 satisfied</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Executes swap, gets cache miss</td>
<td>Completes swap: returns 0 and sets</td>
<td>Exclusive (P2)</td>
<td>Bus/directory services P2 cache miss; generates invalidate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Lock = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>Swap completes and returns 1, and</td>
<td>Enter critical section</td>
<td>Exclusive (P1)</td>
<td>Bus/directory services P1 cache miss; generates write back</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sets Lock = 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Spins, testing if lock = 0</td>
<td></td>
<td></td>
<td>None</td>
</tr>
</tbody>
</table>
# Implementing Spin Locks: Linked Load/Store

<table>
<thead>
<tr>
<th>Lockit:</th>
<th>Operations</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LL</td>
<td>R2,0(R1)</td>
<td>load linked</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R2,lockit</td>
<td>not available - spin</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R2,R0,#1</td>
<td>locked value</td>
</tr>
<tr>
<td>SC</td>
<td>R2,0(R1)</td>
<td>store</td>
</tr>
<tr>
<td>BEQZ</td>
<td>R2,lockit</td>
<td>branch if store fails</td>
</tr>
</tbody>
</table>
MEMORY CONSISTENCY
Coherence vs Consistency

• Coherence defines the behavior of reads and writes to the same memory location
• Consistency defines the behavior of reads and writes with respect to accesses to other memory locations
  ★ we will return to consistency later
• Working assumptions:
  ★ a write does not complete until all processors have seen the effect of that write
  ★ processor does not change the order of a write with respect to any other memory access
Memory Consistency

• Related to accesses to **multiple** shared memory locations
  ★ coherence deals with accesses to a shared location

```
P1:  A = 0;
    ....
    A = 1;
L1:  if (B == 0) ...

P2:  B = 0;
    ....
    B = 1;
L2:  if (A == 0) ...
```
Memory Consistency

- Related to accesses to **multiple** shared memory locations
  - coherence deals with accesses to a shared location

```
P1:   A = 0;
     ....
     A = 1;
L1:   if (B == 0) ...  

P2:   B = 0;
     ....
     B = 1;
L2:   if (A == 0) ...
```

**Sequential Consistency**
Programmer’s View

• Sequential consistency “easy” to reason about
• Most real programs use synchronization
  ★ synchronization primitives usually implemented in libraries
  ★ not using synchronization ⇒ data races
• Allowing sequential consistency for synchronization variables ensure correctness
  ★ can implement relaxed consistency for the rest
Relaxed Consistency Models

• Idea: allow reads and writes to finish out of order, but use synchronization to enforce ordering

• Ways to relax consistency (weak consistency)
  ★ Relaxing $W \rightarrow R$: processor consistency
  ★ Relaxing $W \rightarrow W$: partial store order
  ★ Relaxing $R \rightarrow W$ and $R \rightarrow R$: weak ordering, PowerPC consistency, release consistency

• Two approaches to optimize performance
  ★ use weak consistency, rely on synchronization for correctness
  ★ use sequential or processor consistency with speculative execution
FALLACIES AND PITFALLS
Fallacies and Pitfalls

• Pitfall: Measuring performance of multiprocessors by linear speedup versus execution time
  ★ sequential vs parallel algorithms
  ★ superlinear speedups and cache effects
  ★ strong vs weak scaling

• Pitfall: Not developing software to take advantage of, or optimize for, a multiprocessor architecture