Caches

CSE 410 - Computer Systems
October 24, 2001

Readings and References

• Reading
  – Sections 7.1, 7.2, 7.3, Computer Organization & Design, Patterson and Hennessy

• Other References
  – Chapter 4, Caches for MIPS, See MIPS Run, D. Sweetman

The Quest for Speed - Memory

• If all memory accesses (IF/lw/sw) accessed main memory, programs would run 20 times slower
• And it’s getting worse
  – processors speed up by 50% annually
  – memory accesses speed up by 9% annually
  – it’s becoming harder and harder to keep these processors fed

A Solution: Memory Hierarchy

• Keep copies of the active data in the small, fast, expensive storage
• Keep all data in the big, slow, cheap storage

Memory Hierarchy

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Fabrication Tech</th>
<th>Access Time (ns)</th>
<th>Typ. Size (bytes)</th>
<th>$/MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Registers</td>
<td>&lt;0.5</td>
<td>256</td>
<td>1000</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>SRAM</td>
<td>2</td>
<td>8K</td>
<td>100</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>SRAM</td>
<td>10</td>
<td>1M</td>
<td>100</td>
</tr>
<tr>
<td>Memory</td>
<td>DRAM</td>
<td>50</td>
<td>128M</td>
<td>0.75</td>
</tr>
<tr>
<td>Disk</td>
<td>Magnetic Disk</td>
<td>10M</td>
<td>32G</td>
<td>0.0035</td>
</tr>
</tbody>
</table>

What is a Cache?

• A cache allows for fast accesses to a subset of a larger data store
• Your web browser’s cache gives you fast access to pages you visited recently
  – faster because it’s stored locally
  – subset because the web won’t fit on your disk
• The memory cache gives the processor fast access to memory that it used recently
  – faster because it’s located on the CPU chip
Locality of reference

- Temporal locality - nearness in time
  - Data being accessed now will probably be accessed again soon
  - Useful data tends to continue to be useful
- Spatial locality - nearness in address
  - Data near the data being accessed now will probably be needed soon
  - Useful data is often accessed sequentially

Memory Access Patterns

- Memory accesses don't look like this
  - random accesses
- Memory accesses do look like this
  - hot variables
  - step through arrays

Cache Terminology

- Hit and Miss
  - the data item is in the cache or the data item is not in the cache
- Hit rate and Miss rate
  - the percentage of references that the data item is in the cache or not in the cache
- Hit time and Miss time
  - the time required to access data in the cache (cache access time) and the time required to access data not in the cache (memory access time)

Effective Access Time

\[ t_{\text{effective}} = (h) t_{\text{cache}} + ((1-h) t_{\text{memory}}) \]

aka, Average Memory Access Time (AMAT)

Cache Contents

- When do we put something in the cache?
  - when it is used for the first time
- When do we take something out of the cache?
  - when we need the space in the cache for some other entry
  - all of memory won’t fit on the CPU chip so not every location in memory can be cached
Fully Associative Cache

- In a fully associative cache,
  - any memory word can be placed in any cache line
  - each cache line stores an address and a data value
  - accesses are slow (but not as slow as you would think)

<table>
<thead>
<tr>
<th>Address</th>
<th>Valid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010100</td>
<td>Y</td>
<td>0x000000001</td>
</tr>
<tr>
<td>00001010</td>
<td>N</td>
<td>0x00000100</td>
</tr>
<tr>
<td>01001010</td>
<td>Y</td>
<td>0x00000410</td>
</tr>
<tr>
<td>01011010</td>
<td>Y</td>
<td>0x000012D10</td>
</tr>
<tr>
<td>00011010</td>
<td>N</td>
<td>0x00000005</td>
</tr>
<tr>
<td>11011100</td>
<td>Y</td>
<td>0x0349A291</td>
</tr>
<tr>
<td>01000000</td>
<td>N</td>
<td>0x0000123A8</td>
</tr>
<tr>
<td>11111100</td>
<td>N</td>
<td>0x000000001</td>
</tr>
</tbody>
</table>

Direct Mapped Caches

- Fully associative caches are too slow
- With direct mapped caches the address of the item determines where in the cache to store it
  - In our example, the lower five bits of the address dictate the location of the cache entry
  - The lowest two bits are the byte offset within the word

<table>
<thead>
<tr>
<th>Address</th>
<th>Valid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00010100</td>
<td>Y</td>
<td>0x000000001</td>
</tr>
<tr>
<td>00001010</td>
<td>N</td>
<td>0x00000100</td>
</tr>
<tr>
<td>01001010</td>
<td>Y</td>
<td>0x00000410</td>
</tr>
<tr>
<td>01011010</td>
<td>Y</td>
<td>0x000012D10</td>
</tr>
<tr>
<td>00011010</td>
<td>N</td>
<td>0x00000005</td>
</tr>
<tr>
<td>11011100</td>
<td>Y</td>
<td>0x0349A291</td>
</tr>
<tr>
<td>01000000</td>
<td>N</td>
<td>0x0000123A8</td>
</tr>
<tr>
<td>11111100</td>
<td>N</td>
<td>0x000000001</td>
</tr>
</tbody>
</table>

Address Tags

- A tag is a label for a cache entry indicating where it came from
  - The upper bits of the data item’s address

<table>
<thead>
<tr>
<th>Tag (2)</th>
<th>Index (3)</th>
<th>Byte Offset (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>111</td>
<td>01</td>
</tr>
</tbody>
</table>

Cache with Address Tag

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>11</td>
<td>Y</td>
<td>0x000000001</td>
</tr>
<tr>
<td>00000001</td>
<td>10</td>
<td>N</td>
<td>0x00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>01</td>
<td>Y</td>
<td>0x00000410</td>
</tr>
<tr>
<td>00000011</td>
<td>00</td>
<td>Y</td>
<td>0x000012D10</td>
</tr>
<tr>
<td>00000100</td>
<td>10</td>
<td>N</td>
<td>0x00000005</td>
</tr>
<tr>
<td>00000101</td>
<td>11</td>
<td>Y</td>
<td>0x0349A291</td>
</tr>
<tr>
<td>00000110</td>
<td>00</td>
<td>Y</td>
<td>0x0000123A8</td>
</tr>
<tr>
<td>00000111</td>
<td>10</td>
<td>N</td>
<td>0x000000001</td>
</tr>
</tbody>
</table>

N-way Set Associative Caches

- Direct mapped caches cannot store more than one address with the same index
- If two addresses collide, then you have to kick one of them out
- 2-way associative caches can store two different addresses with the same index
  - 3-way, 4-way and 8-way set associative designs too
- Reduces misses due to conflicts
- Larger sets imply slower accesses
2-way Set Associative Cache

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Valid</th>
<th>Value</th>
<th>Tag</th>
<th>Valid</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>11</td>
<td>Y</td>
<td>0x00000001</td>
<td>00</td>
<td>Y</td>
<td>0x00000002</td>
</tr>
<tr>
<td>001</td>
<td>10</td>
<td>N</td>
<td>0x00000001</td>
<td>10</td>
<td>N</td>
<td>0x00000003</td>
</tr>
<tr>
<td>010</td>
<td>01</td>
<td>Y</td>
<td>0x00000000</td>
<td>11</td>
<td>Y</td>
<td>0x00000004</td>
</tr>
<tr>
<td>011</td>
<td>00</td>
<td>Y</td>
<td>0x00000000</td>
<td>10</td>
<td>N</td>
<td>0x00000005</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>N</td>
<td>0x00000000</td>
<td>11</td>
<td>N</td>
<td>0x00000006</td>
</tr>
<tr>
<td>101</td>
<td>11</td>
<td>Y</td>
<td>0x00000000</td>
<td>10</td>
<td>Y</td>
<td>0x00000007</td>
</tr>
<tr>
<td>110</td>
<td>00</td>
<td>Y</td>
<td>0x00000000</td>
<td>01</td>
<td>Y</td>
<td>0x00000008</td>
</tr>
<tr>
<td>111</td>
<td>10</td>
<td>N</td>
<td>0x00000000</td>
<td>10</td>
<td>N</td>
<td>0x00000009</td>
</tr>
</tbody>
</table>

Spatial Locality

- Using the cache improves performance by taking advantage of temporal locality
  - When a word in memory is accessed it is loaded into cache memory
  - It is then available quickly if it is needed again soon
- This does nothing for spatial locality

Memory Blocks

- Divide memory into blocks
- If any word in a block is accessed, then load an entire block into the cache

Address Tags Revisited

- A cache block size > 1 word requires the address to be divided differently
- Instead of a byte offset into a word, we need a byte offset into the block
- Assuming we had 10-bit addresses, and 4 words in a block...

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>110</td>
<td>0111</td>
</tr>
</tbody>
</table>

The Effects of Block Size

- Big blocks are good
  - Fewer first time misses
  - Exploits spatial locality
- Small blocks are good
  - Don’t evict so much other data when bringing in a new entry
  - More likely that all items in the block will turn out to be useful
Reads vs. Writes

- Caching is essentially making a copy of the data.
- When you read, the copies still match when you’re done.
- When you write, the results must eventually propagate to both copies.
  - Especially at the lowest level, which is in some sense the permanent copy.

Write-Through Caches

- Write all updates to both cache and memory.
- Advantages:
  - The cache and the memory are always consistent.
  - Evicting a cache line is cheap because no data needs to be written out to memory at eviction.
  - Easy to implement.
- Disadvantages:
  - Runs at memory speeds when writing (can use write buffer to reduce this problem).

Write-Back Caches

- Write the update to the cache only. Write to memory only when cache block is evicted.
- Advantage:
  - Runs at cache speed rather than memory speed.
  - Some writes never go all the way to memory.
  - When a whole block is written back, can use high bandwidth transfer.
- Disadvantage:
  - Complexity required to maintain consistency.

Dirty bit

- When evicting a block from a write-back cache, we could:
  - Always write the block back to memory.
  - Write it back only if we changed it.
- Caches use a “dirty bit” to mark if a line was changed:
  - The dirty bit is 0 when the block is loaded.
  - It is set to 1 if the block is modified.
  - When the line is evicted, it is written back only if the dirty bit is 1.

i-Cache and d-Cache

- There usually are two separate caches for instructions and data.
  - Avoids structural hazards in pipelining.
  - The combined cache is twice as big but still has an access time of a small cache.
  - Allows both caches to operate in parallel, for twice the bandwidth.

Cache Line Replacement

- How do you decide which cache block to replace?
- If the cache is direct-mapped, it’s easy:
  - Only one slot per index.
- Otherwise, common strategies:
  - Random.
  - Least Recently Used (LRU).
LRU Implementations

- LRU is very difficult to implement for high degrees of associativity
- 4-way approximation:
  - 1 bit to indicate least recently used pair
  - 1 bit per pair to indicate least recently used item in this pair
- We will see this again at the operating system level

Multi-Level Caches

- Use each level of the memory hierarchy as a cache over the next lowest level
- Inserting level 2 between levels 1 and 3 allows:
  - level 1 to have a higher miss rate (so can be smaller and cheaper)
  - level 3 to have a larger access time (so can be slower and cheaper)

Cache Comparisons

<table>
<thead>
<tr>
<th></th>
<th>L1 i-Cache</th>
<th>L1 d-Cache</th>
<th>L2 unified Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>8KB, direct-mapped</td>
<td>8KB, direct-mapped</td>
<td>96KB, 5-way</td>
</tr>
<tr>
<td></td>
<td>32KB, 2-way (LRU)</td>
<td>32KB, 2-way (LRU)</td>
<td>64B block</td>
</tr>
<tr>
<td></td>
<td>64B block</td>
<td>32B block</td>
<td>on chip</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>MIPS R10000</th>
<th>Pentium Pro</th>
<th>UltraSparc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>8KB, 4-way</td>
<td>8KB, 2-way</td>
<td>16KB, direct-mapped</td>
</tr>
<tr>
<td></td>
<td>32B block</td>
<td>32B block</td>
<td>32B block</td>
</tr>
</tbody>
</table>

Summary: Classifying Caches

- Where can a block be placed?
  - Direct mapped, N-way Set or Fully associative
- How is a block found?
  - Direct mapped: by index
  - Set associative: by index and search
  - Fully associative: by search
- What happens on a write access?
  - Write-back or Write-through
- Which block should be replaced?
  - Random
  - LRU (Least Recently Used)