Memory Subsystem Design

or

Nothing Beats Cold, Hard Cache

Who Cares about Memory Hierarchy?

• Processor Only Thus Far in Course

1980: no cache in µproc;
1995 2-level cache, 60% trans. on Alpha 21164  µproc

Memory Cache

• Can put small, fast memory close to processor.
• What do we put there?

Memory Locality

• Memory hierarchies take advantage of memory locality.
• Memory locality is the principle that future memory accesses are near past accesses.
• Memory hierarchies take advantage of two types of locality
  – Temporal locality -- near in time => we will often access the same data again very soon
  – Spatial locality -- near in space/distance => our next access is often very close to our last access (or recent accesses).

1,2,3,1,2,3,8,8,47,9,10,8,8...
Locality and caching

- Memory hierarchies exploit locality by **caching** (keeping close to the processor) data likely to be used again.
- This is done because we can build large, slow memories and small, fast memories, but we can’t build large, fast memories.
- If it works, we get the illusion of SRAM access time with disk capacity.

SRAM (static RAM) -- 5-20 ns access time, very expensive
DRAM (dynamic RAM) -- 60-100 ns, cheaper
disk -- access time measured in milliseconds, very cheap

---

A typical memory hierarchy

---

Cache Fundamentals

- **hit** -- an access where the data is found in the cache.
- **miss** -- an access which isn’t
- **hit time** -- time to access the higher cache
- **miss penalty** -- time to move data from lower level to upper, then to cpu
- **hit ratio** -- percentage of time the data is found in the higher cache
- **(1 - hit ratio)**

---

Cache Fundamentals, cont.

- **cache block size or cache line size** -- the amount of data that gets transferred on a cache miss.
- **instruction cache** -- cache that only holds instructions.
- **data cache** -- cache that only caches data.
- **unified cache** -- cache that holds both.
A Cache

- blocksize = 4 words, cache size = 2 blocks, 8 words, associativity = full

CPU reads addresses 0, 1, 2, 0, 1, 2, 3, 4, 5, writes 3, reads 2, 5, 7, 14, 5, 15, 3

Cache Organization: Where can a block be placed in the cache?

- Block 12 placed in 8-block cache:
  - Fully associative, direct mapped, n-way set associative
  - Index = pointer to the set in the cache where a memory location might be cached
    (associativity = degree of freedom in placing a particular block of memory)
    (set = a collection of cache blocks with the same cache index)

Cache Access: How Is a Block Found In the Cache?

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>FA</td>
<td>No index, large tags</td>
<td></td>
</tr>
<tr>
<td>DM</td>
<td>Large index, smaller tags</td>
<td></td>
</tr>
</tbody>
</table>
Cache Organization -- Overview

- A typical cache has three dimensions

<table>
<thead>
<tr>
<th>Bytes/block (block size)</th>
<th>Number of sets (can be size)</th>
<th>Blocks/set (associativity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>index</td>
<td>block offset</td>
</tr>
</tbody>
</table>

A set-associative cache

- A cache that can put a line of data in exactly \( n \) places is called \( n \)-way set-associative.
- The cache lines that share the same index are a cache set.

Cache Access

- 16 KB, 4-way set-associative cache, 32-bit address, byte-addressable memory, 32-byte cache blocks/lines
- how many tag bits?
- Where would you find the word at address 0x200356A4?

Which Block Should be Replaced on a Miss?

- Direct Mapped is Easy
- Set associative or fully associative:
  - longest till next use (ideal, impossible)
  - least recently used (best practical approximation)
  - pseudo-LRU (e.g., NMRU, NRU)
  - random (easy)
  - how many bits for LRU?

<table>
<thead>
<tr>
<th>Associativity:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Random</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.18%</td>
<td>5.69%</td>
<td>4.67%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88%</td>
<td>2.01%</td>
<td>1.54%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
What Happens on a Write?

- WT: The information is written to both the block in the cache and to the block in the lower-level memory.
- WB: The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each:
  - WT: read misses cannot result in writes (because of replacements)
  - WB: no writes of repeated writes
- WT always combined with write buffers so that don’t wait for lower level memory

What happens on a write miss?

- write-allocate -- make room for the cache line in the cache, fetch rest of line from memory.
- no-write-allocate (write-around) -- write to lower levels of memory hierarchy, ignoring this cache.

Tradeoffs?

- Which makes most sense for write-back?
- Which makes most sense for write-through?

21264 L1 Data Cache

- 64 KB, 64-byte blocks, 2-way set associative, ? blocks, ? sets
- write-back

Cache Organization:
Separate Instruction and Data Caches?

<table>
<thead>
<tr>
<th>Size</th>
<th>Instruction Cache</th>
<th>Data Cache</th>
<th>Unified Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>3.06%</td>
<td>24.61%</td>
<td>13.34%</td>
</tr>
<tr>
<td>2 KB</td>
<td>2.26%</td>
<td>20.57%</td>
<td>9.78%</td>
</tr>
<tr>
<td>4 KB</td>
<td>1.78%</td>
<td>15.94%</td>
<td>7.24%</td>
</tr>
<tr>
<td>8 KB</td>
<td>1.10%</td>
<td>10.19%</td>
<td>4.57%</td>
</tr>
<tr>
<td>16 KB</td>
<td>0.64%</td>
<td>6.47%</td>
<td>2.87%</td>
</tr>
<tr>
<td>32 KB</td>
<td>0.39%</td>
<td>4.82%</td>
<td>1.99%</td>
</tr>
<tr>
<td>64 KB</td>
<td>0.15%</td>
<td>3.77%</td>
<td>1.35%</td>
</tr>
<tr>
<td>128 KB</td>
<td>0.02%</td>
<td>2.88%</td>
<td>0.95%</td>
</tr>
</tbody>
</table>

if 75% of accesses are instructions?
Other reasons to separate?
Cache Performance

• CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time

• Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)

• Memory stall clock cycles = Memory accesses x Miss rate x Miss penalty

Alternate view of memory performance

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)
Improving Cache Performance

Average memory-access time = Hit time + Miss rate x Miss penalty (ns or clocks)

How are we going to improve cache performance??
1. 
2.
3.

Caches: Key Points

- CPU-Memory gap is a major performance obstacle
- Caches take advantage of program behavior: locality
- Designer has lots of choices -> cache size, block size, associativity, replacement policy, write policy, ...
- Time of program still only reliable performance measure