Caching Basics

CS448

Memory Hierarchies

• Takes advantage of locality of reference principle
  – Most programs do not access all code and data uniformly, but repeat for certain data choices
    • spatial – nearby references are likely
    • temporal – repeat reference is likely
  – Fast memory is expensive per byte
    • Chapter 1 has some of the costs per byte for various forms of memory
  – Make a hierarchy with fast at the top (but not much memory) and slow at the bottom (but lots of memory)
**DRAM/CPU Gap**

CPU performance improves at about 55% per year  
DRAM has only improved at about 7% per year  
• Memory access becoming the bottleneck

**Sample Memory Hierarchy**

- CPU
  - L0 Icache
  - Dcache
  - L1 Cache
- L2 Cache
- Main Memory
  - Disk Cache / RAM Disk
- Disk and I/O Devices, e.g. Network
Hierarchy Concepts

• Data access in terms of blocks at each level
  – Size of a block varies, especially from L0 to L1 to L2, etc.
  – Hit: data you want is found in the cache
  – Miss: data you want is not found in the cache, must be fetched from the lower level memory system
    • Misses cause stall cycles
    • Stall Cycles = IC * Mem-Refs-Per-Instruction * Miss Rate * Miss Penalty

• Four major questions regarding any level in the hierarchy
  – Q1: Where can a block be placed in the upper level?
  – Q2: How is a block found in the upper level?
  – Q3: Which block should be replaced on a miss?
  – Q4: What happens on a write?

Cache Parameters

• Some typical parameters for today’s caches
  • Block / Line Size
    – 16 to 256 bytes
  • Hit time
    – 1 Cycle in L0 Cache
  • Miss penalty
    – 10 to hundreds of clock cycles
  • Access time of next lower memory
    – 4 - 32 clock cycles
  • Miss rate
    – 1% to 20% (application dependent)
  • L0 Cache Size
    – 0 to 512KB
Cache Strategies – Block Placement

• Direct Mapped
  – Each block has only one place to go in a cache, typically
    • Address mod Num-Blocks-In-Cache
    • Usually lower n bits corresponds to the offset in the block, where $2^n = \text{Block size}$, and then another m bits corresponding to the cache block, where $2^m = \text{Num blocks in the cache}$

• Fully Associative
  – Block can be placed anywhere in the cache
  – Implies we must be able to search for it associatively

• Set Associative
  – A set is a group of arbitrary blocks in the cache
  – An address in memory maps to a set, then maps into a block within that set
  – Usually lower n bits corresponds to the offset in the block, where $2^n = \text{Block size}$, and then another m bits corresponding to the set, where $2^m = \text{Num sets}$

Block Identification

• Physical Address
  – For direct-mapped:
    • Tag ## Block-Index ## Block-Offset
  – For set-associative
    • Tag ## Set-Index ## Block-Offset

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>r bits</td>
<td>m bits</td>
<td>n bits</td>
</tr>
</tbody>
</table>

In set-associative cache

$2^m$ sets possible

r bits searched simultaneously for matching block

$2^n$ gives offset to the data into the matching block

–For fully associative?
Typical Cache

- May be composed of 2 types of fast memory devices
  - SRAM’s - hold the actual data and address and status tags in a direct mapped cache
  - TAG RAM’s help with the accounting for set-associative cache
- TAG RAM’s are small associative memories
  - provide fully parallel search capability
  - sequential search would take too long so not even an option
- Where do you have to search
  - fully associative - everywhere
  - set associative - only within the set
  - direct mapped - no search
    - just check for valid and compare one ID

Block Placement

Size of n, m here? Let’s say a block holds 4 bytes
Memory is byte-addressable
Finding a Block

• We’ve already covered how to look up a block in either scheme
  – Find block via direct map or associative mapping, perhaps first finding the right set and then comparing the tag bits for a match
  – Go to the offset of this block to get the memory data
• Extra details
  – Valid bit
    • Added to indicate if a tag entry contains a valid address
  – Dirty bit
    • Added to indicate if data has been written to

Block Replacement

• Random
  – Pick a block at random and discard it
  – For set associative, randomly pick a block within the mapped set
  – Sometimes use pseudo-random instead to get reproducibility at debug time
• LRU - least recently used
  – Need to keep time since each block was last accessed
  – Expensive if number of blocks is large due to global compare
  – Approximation is often used; Use bit tag or counter and LFU
• Replacement strategy critical for small caches
  – doesn’t make a lot of difference for large ones
  – ideal would be a least-likely prediction scheme
  – No simple scheme known for least-likely prediction
Miss Rates

- On VAX traces, block size of 16 bytes

<table>
<thead>
<tr>
<th>Size</th>
<th>Two-way LRU</th>
<th>Two-way Random</th>
<th>Four-way LRU</th>
<th>Four-way Random</th>
<th>Eight-way LRU</th>
<th>Eight-way Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.18%</td>
<td>5.69%</td>
<td>4.67%</td>
<td>5.29%</td>
<td>4.39%</td>
<td>4.96%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.88%</td>
<td>2.01%</td>
<td>1.54%</td>
<td>1.66%</td>
<td>1.39%</td>
<td>1.53%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
<td>1.13%</td>
<td>1.12%</td>
<td>1.12%</td>
</tr>
</tbody>
</table>

- Not much difference with larger caches
- Not much difference with eight-way scheme

Cache Writes

- Reads dominate cache access
  - Only 7% of overall memory traffic are writes on DLX
  - Implies we should make reads fastest (which is good because it is easier to handle)
    - Can read block the same time we compare the tag for a match; ignore value if there is no match
    - Can’t do this with writes, if we compare the tag and write at the same time, we’d have to undo the write if there was no match
      - So we wait an extra cycle for result of the tag comparison
      - Complicated more if we write multiple bytes
Cache Write Schemes

• Write Through
  – Write information back to the cache and to the lower-level memory at the same time
    • Lower level access is slower, though
  – Maintains memory consistency for other devices
    • Other processors
    • DMA

• Write Back
  – Write to cache only, but set the dirty bit when we write
  – Dirty blocks are written back to memory only when replaced
  – Faster, independent of main memory speeds
    • But causes complications with memory consistency

Write Stalls

• Occur when we must stall for a write to complete

• Write miss may generate write stall
  – Write Buffers allows processor to act as soon as data written to the buffer, providing overlapping execution with memory
    • decrease write stalls but do not eliminate them

• Common operations on a write miss
  – write allocate
    • load the block, do the write
    • usually the choice for write back caches so the data is available for a subsequent read or writes
  – No-write allocate or write around
    • modify the block in the lower-level memory, do not load the block in the cache
    • usually the choice for write through caches since subsequent writes would go to memory anyway
Example – Alpha AXP 21064

- Cache – 8192 bytes
- 32 byte block size
- Direct-Mapped Cache
  - Four block write buffer
  - No-write allocate on a write miss
- 34 bit physical address
  - Need 5 bits for the block offset
  - $8192 / 32 = 256$ cache blocks, need 8 bits for block number
  - Remaining 21 bits for a tag

Alpha AXP 21064 Cache
Improving Write Performance

- Data writes can contain four 64-bit words
- CPU can continue to work while the write buffer writes to memory
- **Write Merging**
  - Check to see if other modified blocks could be written as well, i.e sequential. If so, write them too

---

Improving Cache Performance

- Processor needs data and instructions
- Two separate caches implemented for Alpha
  - Avoids structural hazard problems with fetching instruction, data we discussed with pipelining
  - The two caches have separate access patterns
- When there is one cache containing both, it is called a unified or mixed cache
Miss Rates for Caches

SPEC92 on an AXP 21064 DECstation 5000

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

What is missing from this chart?

Miss Rates for Caches (2)

We need to know the % of time we go to the I-Cache and D-Cache
100% of I-Cache accesses, 26% Load, 9% Store
100/(100+26+9) = 0.74 for I-Cache, 0.26 for D-Cache

<table>
<thead>
<tr>
<th>Size</th>
<th>I-Cache</th>
<th>D-Cache</th>
<th>Ave</th>
<th>Unified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>3.06%</td>
<td>24.61%</td>
<td>8.66%</td>
<td>13.34%</td>
</tr>
<tr>
<td>2K</td>
<td>2.26%</td>
<td>20.57%</td>
<td>7.02%</td>
<td>9.78%</td>
</tr>
<tr>
<td>4K</td>
<td>1.78%</td>
<td>15.94%</td>
<td>5.46%</td>
<td>7.24%</td>
</tr>
<tr>
<td>8K</td>
<td>1.10%</td>
<td>10.19%</td>
<td>3.46%</td>
<td>4.57%</td>
</tr>
<tr>
<td>16K</td>
<td>0.64%</td>
<td>6.47%</td>
<td>2.15%</td>
<td>2.87%</td>
</tr>
<tr>
<td>32K</td>
<td>0.39%</td>
<td>4.82%</td>
<td>1.54%</td>
<td>1.99%</td>
</tr>
<tr>
<td>64K</td>
<td>0.15%</td>
<td>3.77%</td>
<td>1.09%</td>
<td>1.35%</td>
</tr>
<tr>
<td>128K</td>
<td>0.02%</td>
<td>2.88%</td>
<td>0.76%</td>
<td>0.95%</td>
</tr>
</tbody>
</table>
Cache Performance

- Average memory access time = Hit time + (Miss Rate * Miss Penalty)
- Memory Stall Clock Cycles = (Reads * Read Miss Rate * Read Penalty) + (Writes * Write Miss Rate * Write Penalty)
- Sometimes we combine reads and writes as an approximation using a generic “Miss”
  - Memory Stall Clock Cycles = Memory Accesses * Miss Rate * Miss Penalty
- CPU Time = IC * (CPI_{exec} + Mem Accesses/Instr * Miss Rate * Miss Penalty) * Clock Cycle Time

Cache Performance Example

Cache Miss Penalty = 50 cycles
Instructions normally take 2 cycles, ignoring stalls
Cache Miss rate is 2%
Average of 1.33 Memory References per instruction

Impact of cache vs. no cache?

CPU Time = IC * (CPI_{exec} + Mem Accesses/Instr * Miss Rate * Miss Penalty) * Clock Cycle Time

CPU Time(cache) = IC * (2 + 1.33 * 0.02 * 50) * Cycle Time
= 3.33 * IC * Cycle Time ; from 2 to 3.33 when not perfect
CPU Time(nocache) = IC * (2 + 1.33 * 50) * Cycle Time
= 68.5 ; Over 30 times longer!
Cache Performance Limitations

- Caches can have a huge impact on performance
- **Downside**
  - The lower the CPI(exec) the higher the relative impact of a fixed number of cache miss clock cycles
  - CPU’s with higher clock rates and same memory hierarchy has a larger number of clock cycles per miss
- **Bottom line**: Amdahl’s Law strikes again, impact of caching can slow us down as we get high clock rates
- Set-Associative Cache appears to perform best on the simulation data
  - Implementing set-associative cache requires some extra multiplexing to select the block we want in the set
  - Increases basic cycle time the larger each set is
  - Book example: This basic cycle time could make set-associative caches slower than direct-mapped caches in some cases!

Sources of Cache Misses – 3 C’s

- **Compulsory**
  - first access to a block, no choice but to load it
  - also called cold-start or first-reference misses
- **Capacity**
  - if working set is too big for the cache then even after steady state they all won’t fit
  - Therefore, needed lines will be displaced by other needed lines
  - thrashing possible if we later want to retrieve something tossed
- **Conflict**
  - Collision as a result of the block placement strategy
  - Data you want maps to the same block in the cache
- **Data on these three miss types for 1-8 way caches**
<table>
<thead>
<tr>
<th>Cache size</th>
<th>Degree associative</th>
<th>Total miss rate</th>
<th>Compulsory</th>
<th>Capacity</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 KB</td>
<td>1-way</td>
<td>0.133</td>
<td>0.002</td>
<td>1%</td>
<td>0.080</td>
</tr>
<tr>
<td>1 KB</td>
<td>2-way</td>
<td>0.105</td>
<td>0.002</td>
<td>2%</td>
<td>0.080</td>
</tr>
<tr>
<td>1 KB</td>
<td>4-way</td>
<td>0.095</td>
<td>0.002</td>
<td>2%</td>
<td>0.080</td>
</tr>
<tr>
<td>1 KB</td>
<td>8-way</td>
<td>0.087</td>
<td>0.002</td>
<td>2%</td>
<td>0.080</td>
</tr>
<tr>
<td>2 KB</td>
<td>1-way</td>
<td>0.098</td>
<td>0.002</td>
<td>2%</td>
<td>0.044</td>
</tr>
<tr>
<td>2 KB</td>
<td>2-way</td>
<td>0.076</td>
<td>0.002</td>
<td>2%</td>
<td>0.044</td>
</tr>
<tr>
<td>2 KB</td>
<td>4-way</td>
<td>0.064</td>
<td>0.002</td>
<td>3%</td>
<td>0.044</td>
</tr>
<tr>
<td>2 KB</td>
<td>8-way</td>
<td>0.054</td>
<td>0.002</td>
<td>4%</td>
<td>0.044</td>
</tr>
<tr>
<td>4 KB</td>
<td>1-way</td>
<td>0.072</td>
<td>0.002</td>
<td>3%</td>
<td>0.031</td>
</tr>
<tr>
<td>4 KB</td>
<td>2-way</td>
<td>0.057</td>
<td>0.002</td>
<td>3%</td>
<td>0.031</td>
</tr>
<tr>
<td>4 KB</td>
<td>4-way</td>
<td>0.049</td>
<td>0.002</td>
<td>4%</td>
<td>0.031</td>
</tr>
<tr>
<td>4 KB</td>
<td>8-way</td>
<td>0.039</td>
<td>0.002</td>
<td>5%</td>
<td>0.031</td>
</tr>
<tr>
<td>8 KB</td>
<td>1-way</td>
<td>0.046</td>
<td>0.002</td>
<td>4%</td>
<td>0.023</td>
</tr>
<tr>
<td>8 KB</td>
<td>2-way</td>
<td>0.038</td>
<td>0.002</td>
<td>5%</td>
<td>0.023</td>
</tr>
<tr>
<td>8 KB</td>
<td>4-way</td>
<td>0.035</td>
<td>0.002</td>
<td>5%</td>
<td>0.023</td>
</tr>
<tr>
<td>8 KB</td>
<td>8-way</td>
<td>0.029</td>
<td>0.002</td>
<td>6%</td>
<td>0.023</td>
</tr>
<tr>
<td>16 KB</td>
<td>1-way</td>
<td>0.029</td>
<td>0.002</td>
<td>7%</td>
<td>0.015</td>
</tr>
<tr>
<td>16 KB</td>
<td>2-way</td>
<td>0.022</td>
<td>0.002</td>
<td>9%</td>
<td>0.015</td>
</tr>
<tr>
<td>16 KB</td>
<td>4-way</td>
<td>0.020</td>
<td>0.002</td>
<td>10%</td>
<td>0.015</td>
</tr>
<tr>
<td>16 KB</td>
<td>8-way</td>
<td>0.018</td>
<td>0.002</td>
<td>10%</td>
<td>0.015</td>
</tr>
</tbody>
</table>

## Miss Rates

<table>
<thead>
<tr>
<th>Cache size</th>
<th>Degree associative</th>
<th>Total miss rate</th>
<th>Compulsory</th>
<th>Capacity</th>
<th>Conflict</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KB</td>
<td>1-way</td>
<td>0.020</td>
<td>0.002</td>
<td>10%</td>
<td>0.010</td>
</tr>
<tr>
<td>32 KB</td>
<td>2-way</td>
<td>0.014</td>
<td>0.002</td>
<td>14%</td>
<td>0.010</td>
</tr>
<tr>
<td>32 KB</td>
<td>4-way</td>
<td>0.013</td>
<td>0.002</td>
<td>15%</td>
<td>0.010</td>
</tr>
<tr>
<td>32 KB</td>
<td>8-way</td>
<td>0.013</td>
<td>0.002</td>
<td>15%</td>
<td>0.010</td>
</tr>
<tr>
<td>64 KB</td>
<td>1-way</td>
<td>0.014</td>
<td>0.002</td>
<td>14%</td>
<td>0.007</td>
</tr>
<tr>
<td>64 KB</td>
<td>2-way</td>
<td>0.010</td>
<td>0.002</td>
<td>20%</td>
<td>0.007</td>
</tr>
<tr>
<td>64 KB</td>
<td>4-way</td>
<td>0.009</td>
<td>0.002</td>
<td>21%</td>
<td>0.007</td>
</tr>
<tr>
<td>64 KB</td>
<td>8-way</td>
<td>0.009</td>
<td>0.002</td>
<td>22%</td>
<td>0.007</td>
</tr>
<tr>
<td>128 KB</td>
<td>1-way</td>
<td>0.010</td>
<td>0.002</td>
<td>20%</td>
<td>0.004</td>
</tr>
<tr>
<td>128 KB</td>
<td>2-way</td>
<td>0.007</td>
<td>0.002</td>
<td>29%</td>
<td>0.004</td>
</tr>
<tr>
<td>128 KB</td>
<td>4-way</td>
<td>0.006</td>
<td>0.002</td>
<td>31%</td>
<td>0.004</td>
</tr>
<tr>
<td>128 KB</td>
<td>8-way</td>
<td>0.006</td>
<td>0.002</td>
<td>31%</td>
<td>0.004</td>
</tr>
</tbody>
</table>
Reducing Cache Misses

- We’ll examine seven techniques
  - Larger Block Size
  - Higher Associativity
  - Victim Caches
  - Pseudo-Associative Caches
  - Hardware prefetching
  - Compiler prefetching
  - Compiler Optimizations

We’ll cover these the next lecture
#1: Increased Block Size

- Advantages
  - Reduce compulsory misses due to spatial locality

- Disadvantages
  - Larger block takes longer to move, so higher penalty for miss
  - More conflicts now though, because there are fewer blocks in the cache, so more memory blocks map to the same cache blocks

Miss Rate vs. Block Size

![Graph showing Miss Rate vs. Block Size](image-url)

- Sweet Spot
- Cache Size

---

31

32
#2: Higher Associativity

- **2:1 Rule of Thumb**
  - 2 way set associative cache of size $N/2$ is about the same as a direct mapped cache of size $N$
  - So does this mean even more associations is better?

- **Advantage**
  - Higher associativity should reduce conflicts

- **Disadvantage**
  - Higher associativity can reduce number of sets, if we keep the same cache size
  - There is overhead with higher associativity in the hardware, increases the basic clock cycle for all instructions

## Associativity Example

Assume higher associativity increases the clock cycle as:
- $\text{CycleTime}(2\text{-way}) = 1.10 \times \text{CycleTime}(1\text{-way})$
- $\text{CycleTime}(4\text{-way}) = 1.12 \times \text{CycleTime}(1\text{-way})$
- $\text{CycleTime}(8\text{-way}) = 1.14 \times \text{CycleTime}(1\text{-way})$

A hit takes 1 cycle, miss penalty for direct-map is 50 cycles

Calculate Ave. Mem Access Times

Ave mem Access Time = HitTime + miss Rate * Miss Penalty

AveTime(1-way) = 1 + MissRate(1-way) * 50
AveTime(2-way) = 1.10 * MissRate(2-way) * 50

(continued on next slide)
Associativity Example (2)

Look up miss rates for the different caches from the previous table e.g. for direct-mapped cache, miss rate = 0.133 at 1Kb cache

\[
\text{AveTime(1-way)} = 1 + 0.133 \times 50 = 7.65
\]

... 

Gives us the table below:

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>One-way</th>
<th>Two-way</th>
<th>Four-way</th>
<th>Eight-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.65</td>
<td>6.60</td>
<td>6.22</td>
<td>5.44</td>
</tr>
<tr>
<td>2</td>
<td>5.90</td>
<td>4.90</td>
<td>4.62</td>
<td>4.09</td>
</tr>
<tr>
<td>4</td>
<td>4.60</td>
<td>3.95</td>
<td>3.57</td>
<td>3.19</td>
</tr>
<tr>
<td>8</td>
<td>3.30</td>
<td>3.00</td>
<td>2.87</td>
<td>2.59</td>
</tr>
<tr>
<td>16</td>
<td>2.45</td>
<td>2.20</td>
<td>2.12</td>
<td>2.04</td>
</tr>
<tr>
<td>32</td>
<td>2.00</td>
<td>1.80</td>
<td>1.77</td>
<td>1.79</td>
</tr>
<tr>
<td>64</td>
<td>1.70</td>
<td>1.60</td>
<td>1.57</td>
<td>1.59</td>
</tr>
<tr>
<td>128</td>
<td>1.50</td>
<td>1.45</td>
<td>1.42</td>
<td>1.44</td>
</tr>
</tbody>
</table>

Yikes!

#3: Victim Caches

- Idea: with direct mapping, conflict only occurs with a small number of blocks
  - Might occur frequently, but with not too many blocks
  - Very bad if we thrash among these direct mapped blocks to the same cache block
  - So use a small, fully-associative cache to store what was thrown out

- Add a small “victim” cache between the main cache and main memory
  - This cache only stores data discarded from a cache miss
  - Keep it small, so it is easy to implement, even associatively
  - If data is not in the main cache but is in the victim cache, swap the data from the main/victim cache
  - Since we address the victim cache and main cache at the same time, there is no increased penalty with this scheme
Victim Caches

• Study by Jouppi
  – Victim cache of 1-5 entries effective at reducing conflict misses for small, direct-mapped caches
  – Removed 20-95% of conflict misses in a 4K direct mapped cache
    • Of course this is a very small cache by today’s standards
    • Not as much benefit with larger caches, even if direct-mapped, due to alleviation of conflicts