EECS 252 Graduate Computer Architecture

Lec 12 - Caches

David Culler
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~culler
http://www-inst.eecs.berkeley.edu/~cs252

Review: Who Cares About the Memory Hierarchy?
- Processor Only Thus Far in Course:
  - CPU cost/performace, ISA, Pipelined Execution

μProc

60%/yr.

CPU-DRAM Gap

"Moore's Law"

7%/yr.

"Less' Law?"


Review: What is a cache?
- Small, fast storage used to improve average access time to slow memory.
- Exploits spacial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers a cache on variables
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?

Bigger 

Proc/Regs 

L1-Cache 

Faster 

Memory

L2-Cache

Disk, Tape, etc.

Proc/Regs

L1-Cache

L2-Cache

Memory

Disk, Tape, etc.

Why it works
- Exploit the statistical properties of programs
- Locality of reference
  - Temporal
  - Spatial

Why it works

Locality of reference

- Temporal
- Spatial

Average Memory Access Time

AMAT = HitTime × MissRate × MissPenalty

HitTime = HitRate × MissRate × MissPenalty

HitTime = HitRate × MissRate × MissPenalty

HitTime = HitRate × MissRate × MissPenalty

Simple hardware structure that observes program behavior and reacts to improve future performance
- Is the cache visible in the ISA?

Block Placement

Q1: Where can a block be placed in the upper level?
  - Fully Associative
  - Set Associative
  - Direct Mapped
1 KB Direct Mapped Cache, 32B blocks

- For a $2^N$ byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = $2^M$)

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Example: 0x50</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stored as part of the cache &quot;state&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Valid Bit</th>
<th>Cache Tag</th>
<th>Ex: 0x50</th>
<th>Ex: 0x50</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 31</td>
<td>Byte 31</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 1</td>
<td>Byte 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Byte 32</td>
<td>Byte 32</td>
</tr>
</tbody>
</table>

Review: Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
  - How big is the tag?

Example: Two-way set associative cache
- Cache Index selects a "set" from the cache
- The two tags in the set are compared to the input in parallel
- Data is selected based on the tag result

<table>
<thead>
<tr>
<th>Cache Index</th>
<th>Cache Tag</th>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x50</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x50</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0x50</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x50</td>
<td></td>
</tr>
</tbody>
</table>

Q2: How is a block found if it is in the upper level?

- Index identifies set of possibilities
- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, expands tag

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>

Cache size = Associativity * $2^{index\_size} * 2^{offset\_size}$

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Assoc: 2-way 4-way 8-way</th>
<th>Size</th>
<th>LRU</th>
<th>Ran</th>
<th>LRU</th>
<th>Ran</th>
<th>LRU</th>
<th>Ran</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
<td>5.3%</td>
<td>4.4%</td>
<td>5.0%</td>
<td></td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
<td>1.7%</td>
<td>1.4%</td>
<td>1.5%</td>
<td></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>
<td></td>
</tr>
</tbody>
</table>

Q4: What happens on a write?

- Write through—The information is written to both the block in the cache and to the block in the lower-level memory.
- Write back—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
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don’t wait for lower level memory
- What about on a miss?
  - Write_no_allocate vs write_allocate

Write Buffer for Write Through

- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
Review: Cache performance

- Miss-oriented Approach to Memory Access:
  \[ \text{CPU time} = \text{IC} \times \left( \frac{\text{CPI}_{\text{total}} \times \text{Miss rate} \times \text{Miss penalty}}{\text{Instruction}} \right) \times \text{Cycle time} \]

- Separating out Memory component entirely
  \[ \text{AMAT} = \left( \frac{\text{Mem Access}}{\text{Instruction}} \right) \times \text{AMAT} \times \text{Cycle time} \]

  \[ \text{Effective CPI} = \text{CPI}_{\text{ideal mem}} + \text{P}_{\text{mem}} \times \text{AMAT} \]

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control

- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty

- CPI = ideal CPI + average stalls per instruction
  \[ 1.1(\text{cycles/ins}) + [0.30(\text{DataMops/ins}) \times 0.10(\text{miss/DataMop}) \times 50(\text{cycle/miss})] + [1(\text{InstMops/ins}) \times 0.01(\text{miss/InstMop}) \times 50(\text{cycle/miss})] = (1.1 + 1.5 + 0.5) \text{cycles/ins} = 3.1 \]

- 58% of the time the proc is stalled waiting for memory!

- AMAT = \( \frac{75\% \times (1+0.64\times50)+25\% \times (1+6.47\times50)}{1+0.01 \times 50} = 2.05 \)

- AMAT = \( \frac{75\% \times (1+1.99\times50)+25\% \times (1+1.99\times50)}{1+0.1 \times 50} = 2.24 \)

Example: Harvard Architecture

- Unified vs Separate I&D (Harvard)

  - Statistics (given in H&P):
    - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
    - 32KB unified: Aggregate miss rate=1.99%

- Which is better (ignore L2 cache)?
  - Assume 33% data cps = 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that data hit has 1 stall for unified cache (only one port)

  \[ \text{AMAT}_{\text{Harvard}} = \frac{75\% \times (1+0.64\times50)+25\% \times (1+6.47\times50)}{1+0.01 \times 50} = 2.05 \]

  \[ \text{AMAT}_{\text{Unified}} = \frac{75\% \times (1+1.99\times50)+25\% \times (1+1.99\times50)}{1+0.1 \times 50} = 2.24 \]

The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back

- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
    - depends on technology / cost

- Simplicity often wins

Review: Improving Cache Performance

\[ \text{CPU time} = \text{IC} \times \left( \frac{\text{CPI}_{\text{total}} \times \text{Miss rate} \times \text{Miss penalty}}{\text{Instruction}} \right) \times \text{Clock cycle time} \]

1. Reduce the miss rate.
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

Reducing Misses

- Classifying Misses: 3 Cs

  - **Compulsory**—The first access to a block is not in the cache, so the block must be brought into the cache. Also called cold start misses or first reference misses. (Misses in even an Infinite Cache)

  - **Capacity**—If the cache cannot contain all the blocks needed during execution of a program, capacity misses will occur due to blocks being discarded and later retrieved. (Misses in Fully Associative Size X Cache)

  - **Conflict**—If block-placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called collision misses or interference misses. (Misses in N-way Associative, Size X Cache)

- More recent, 4th “C”:
  - **Coherence**—Misses caused by cache coherence.
### 3Cs Absolute Miss Rate (SPEC92)

![Graph]

- **Compulsory vanishingly small**
- **Conflict**
- **Capacity**

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.14</td>
</tr>
<tr>
<td>2</td>
<td>0.12</td>
</tr>
<tr>
<td>4</td>
<td>0.10</td>
</tr>
<tr>
<td>8</td>
<td>0.08</td>
</tr>
<tr>
<td>16</td>
<td>0.06</td>
</tr>
<tr>
<td>32</td>
<td>0.04</td>
</tr>
<tr>
<td>64</td>
<td>0.02</td>
</tr>
</tbody>
</table>

### 2:1 Cache Rule

*Miss rate 1-way associative cache size X ~ miss rate 2-way associative cache size X/2*

![Graph]

### How Can Reduce Misses?

- **3 Cs**: Compulsory, Capacity, Conflict
- In all cases, assume total cache size not changed:
- What happens if:
  1. Change Block Size:
     Which of 3Cs is obviously affected?
  2. Change Associativity:
     Which of 3Cs is obviously affected?
  3. Change Algorithm / Compiler:
     Which of 3Cs is obviously affected?

### 3Cs Relative Miss Rate

![Graph]

- **Conflict**
- **Capacity**

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>80%</td>
</tr>
<tr>
<td>4</td>
<td>60%</td>
</tr>
<tr>
<td>8</td>
<td>40%</td>
</tr>
<tr>
<td>16</td>
<td>20%</td>
</tr>
<tr>
<td>32</td>
<td>0%</td>
</tr>
</tbody>
</table>

Caveat: fixed block size

### 1. Reduce Misses via Larger Block Size

- **1K**
- **4K**
- **16K**
- **64K**
- **256K**

<table>
<thead>
<tr>
<th>Block Size (bytes)</th>
<th>Miss Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>25%</td>
</tr>
<tr>
<td>32</td>
<td>15%</td>
</tr>
<tr>
<td>64</td>
<td>10%</td>
</tr>
<tr>
<td>128</td>
<td>5%</td>
</tr>
<tr>
<td>256</td>
<td>0%</td>
</tr>
</tbody>
</table>

### 2. Reduce Misses via Higher Associativity

- **2:1 Cache Rule**:
  - Miss Rate DM cache size N ~ Miss Rate 2-way cache size N/2
- Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%
Example: Avg. Memory Access Time vs. Miss Rate

- assume CCT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

<table>
<thead>
<tr>
<th>KB</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.33</td>
<td>2.15</td>
<td>2.07</td>
<td>2.01</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.86</td>
<td>1.76</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>1.72</td>
<td>1.67</td>
<td>1.61</td>
<td>1.53</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
<td>1.40</td>
<td>1.47</td>
<td>1.43</td>
</tr>
<tr>
<td>16</td>
<td>1.29</td>
<td>1.22</td>
<td>1.22</td>
<td>1.22</td>
</tr>
<tr>
<td>32</td>
<td>1.20</td>
<td>1.24</td>
<td>1.26</td>
<td>1.27</td>
</tr>
<tr>
<td>64</td>
<td>1.14</td>
<td>1.20</td>
<td>1.21</td>
<td>1.23</td>
</tr>
<tr>
<td>128</td>
<td>1.10</td>
<td>1.17</td>
<td>1.18</td>
<td>1.20</td>
</tr>
</tbody>
</table>

(Red means A.M.A.T. not improved by more associativity)

3. Reducing Misses via a “Victim Cache”

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines

4. Reducing Misses via “Pseudo-Associativity”

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?
- Divide cache: on a miss, check other half of cache to see if there, if so have a pseudo-hit (slow hit)

<table>
<thead>
<tr>
<th>Hit Time</th>
<th>Pseudo Hit Time</th>
<th>Miss Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time</td>
<td>Time</td>
</tr>
</tbody>
</table>

- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
  - Better for caches not tied directly to processor (L2)
  - Used in MIPS R1000 L2 cache, similar in UltraSPARC

5. Reducing Misses by Hardware Prefetching of Instructions & Data

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in “stream buffer”
  - On miss check stream buffer
- Works with data blocks too:
  - Jouppi [1990]: 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%  
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches
- Prefetching relies on having extra memory bandwidth that can be used without penalty

6. Reducing Misses by Software Prefetching Data

- Data Prefetch
  - Load data into register (HP PA-RISC loads)
  - Cache Prefetch: load into cache
    (MIPS IV, PowerPC, SPARC v. 9)
  - Special prefetching instructions cannot cause faults; a form of speculative execution
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

7. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts (using tools they developed)
- Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows

NOW Handout Page 5
Merging Arrays Example

/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];

Reducing conflicts between val & key; improve spatial locality

Loop Interchange Example

/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 500; i = i+1)
      x[i][j] = 2 * x[i][j];
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 500; i = i+1)
    for (j = 0; j < 100; j = j+1)
      x[i][j] = 2 * x[i][j];

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Loop Fusion Example

/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    { a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];
    }

2 misses per access to a & c vs. one miss per access; improve spatial locality

Blocking Example

/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    r = 0;
    for (k = 0; k < N; k = k+1)
      r = r + y[i][k]*z[k][j];
  x[i][j] = r;
/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B-1,N); j = j+1)
        { r = 0;
          for (k = kk; k < min(kk+B-1,N); k = k+1)
            r = r + y[i][k]*z[k][j];
          x[i][j] = r;
        }

• Two Inner Loops:
  - Read all N*N elements of y
  - Read N elements of 1 row of y repeatedly
  - Write N elements of 1 row of x

• Capacity Misses a function of N & Cache Size:
  - 2*N*N => (assuming no conflict; otherwise …)
• Idea: compute on BxB submatrix that fits

Reducing Conflict Misses by Blocking

• Conflict misses in caches not FA vs. Blocking size
  - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs.
    48 despite both fit in cache

Blocking Factor

0 0.05 0.1
0 50 100 150

Fully Associative Cache

Direct Mapped Cache

Now Handout Page 6
Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)

- merged arrays
- loop interchange
- loop fusion
- blocking

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

Impact of Memory Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?
- Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
- Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000
Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.

Review: Four Questions for Memory Hierarchy Designers

- Q1: Where can a block be placed in the upper level? 
  (Block placement)
  - Fully Associative, Set Associative, Direct Mapped
- Q2: How is a block found if it is in the upper level? 
  (Block identification)
  - Tag/Block
- Q3: Which block should be replaced on a miss? 
  (Block replacement)
  - Random, LRU
- Q4: What happens on a write? 
  (Write strategy)
  - Write Back or Write Through (with Write Buffer)

Summary

- 3 Cs: Compulsory, Capacity, Conflict
  1. Reduce Misses via Larger Block Size
  2. Reduce Misses via Higher Associativity
  3. Reducing Misses via Victim Cache
  4. Reducing Misses via Pseudo-Associativity
  5. Reducing Misses by HW Prefetching Instr, Data
  6. Reducing Misses by SW Prefetching Data
  7. Reducing Misses by Compiler Optimizations

Remember danger of concentrating on just one parameter when evaluating performance