Black Hat (a security conference) is currently underway, and this year there are several talks focusing on automatically identifying and classifying software bugs. Noah Johnson (our TA) is currently at this conference giving a talk on BitBlaze, one of the tools discussed in this article. Check it out.

And in Review...

- We would like to have the capacity of disk at the speed of the processor: unfortunately this is not feasible.
- So we create a memory hierarchy:
  - each successively lower level contains “most used” data from next higher level
  - exploits temporal & spatial locality
  - do the common case fast, worry less about the exceptions (design principle of MIPS)
- Locality of reference is a Big Idea

**Where we left off: direct mapped cache example**

**Ex.: 16KB of data, direct-mapped, 4 word blocks**

<table>
<thead>
<tr>
<th>Offset: 4 bits</th>
<th>Address (hex)</th>
<th>Value of Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000010</td>
<td>a</td>
</tr>
<tr>
<td>1</td>
<td>00000014</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>00000018</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>0000001C</td>
<td>d</td>
</tr>
</tbody>
</table>

**Index: 10 bits**

<table>
<thead>
<tr>
<th>Index: 18 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000010</td>
</tr>
<tr>
<td>000000014</td>
</tr>
<tr>
<td>000000018</td>
</tr>
<tr>
<td>00000001C</td>
</tr>
</tbody>
</table>

**Read 4 addresses**

1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x0000003C

**Memory vals here:**

<table>
<thead>
<tr>
<th>Memory vals here</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00000010</td>
</tr>
<tr>
<td></td>
<td>00000014</td>
</tr>
<tr>
<td></td>
<td>00000018</td>
</tr>
<tr>
<td></td>
<td>0000001C</td>
</tr>
</tbody>
</table>

**3. Load Word from 0x00000034**

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag field</th>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xc-f</td>
<td>0x8-b</td>
<td>0x4-7</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>c</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Valid**

<table>
<thead>
<tr>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

**No valid data**

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xc-f</td>
<td>0x8-b</td>
</tr>
<tr>
<td>1</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Valid**

<table>
<thead>
<tr>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

So read block 3
Load that cache block, return word f

- 0000000000000000 000000011 0100
- Valid
- Index
- Tag field
- Offset
- 0
- 0
- 0
- d
cb
a
-
-
-
-
-
0
1
2
3
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
Do an example yourself. What happens?

• Chose from: Cache: Hit, Miss, Miss w. replace
  Values returned: a, b, c, d, e, ..., k, l
• Load word from address 0x00000030 ?
  000000000000000000 0000000011 0000
• Load word from address 0x00000001c ?
  000000000000000000 0000000001 1100

Valid
Tag 0xc-f 0x8-b 0x4-7 0x0-3
Index

What to do on a cache write?

• First, we need to get data into the cache
• Follow same procedure as a read. If a cache miss, read from memory. Once there, we have to make a policy decision.
• Write-through
  • update the word in cache block and corresponding word in memory
• Write-back
  • update word in cache block
  • allow memory word to be “stale”
  • add ‘dirty’ bit to each block indicating that memory needs to be updated when block is replaced
  • OS flushes cache before I/O...

Performance trade-offs?

Block Size Tradeoff (1/3)

• Benefits of Larger Block Size
  • Spatial Locality: if we access a given word, we’re likely to access other nearby words soon
  • Very applicable with Stored-Program Concept: if we execute a given instruction, it’s likely that we’ll execute the next few as well
  • Works nicely in sequential array accesses too

Block Size Tradeoff (2/3)

• Drawbacks of Larger Block Size
  • Larger block size means larger miss penalty
    • on a miss, takes longer time to load a new block from next level
  • If block size is too big relative to cache size, then there are too few blocks
    • Ping-pong effect, compromises temporal locality
  • Result: miss rate goes up
• In general, minimize Average Memory Access Time (AMAT)
  = Hit Time + Miss Penalty x Miss Rate

Block Size Tradeoff (3/3)

• Hit Time
  • time to find and retrieve data from current level cache
• Miss Penalty
  • average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)
• Hit Rate
  • % of requests that are found in current level cache
• Miss Rate
  • 1 - Hit Rate

Block Size Tradeoff Conclusions

For a fixed cache size:

Exploits Spatial Locality
Fewer blocks: compromises temporal locality

Increased Miss Penalty & Miss Rate
Administrivia

- Drop deadline Friday.
  - But you shouldn’t drop. That would make me sad. =(
- My office hours are in 551 Soda today, not 511 Soda.
  - 551 is directly across the hall from 511. Just turn right instead of left when leaving elevator.
- Midterm regrades returned in lab today

Types of Cache Misses (1/2)

- “Three Cs” Model of Misses
- 1st C: Compulsory Misses
  - occur when a program is first started
  - cache does not contain any of that program’s data yet, so misses are bound to occur
  - can’t be avoided easily, so won’t focus on these in this course

Types of Cache Misses (2/2)

- 2nd C: Conflict Misses
  - miss that occurs because two distinct memory addresses map to the same cache location
  - two blocks (which happen to map to the same location) can keep overwriting each other
  - big problem in direct-mapped caches
  - how do we lessen the effect of these?
- Dealing with Conflict Misses
  - Solution 1: Make the cache size bigger
    - Fails at some point
  - Solution 2: Multiple distinct blocks can fit in the same cache Index?

Fully Associative Cache (1/3)

- Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: non-existant
- What does this mean?
  - any block can go anywhere in the cache
  - must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

- Fully Associative Cache (e.g., 32 B block)
  - compare tags in parallel
  - 

<table>
<thead>
<tr>
<th>Tag (27 bits long)</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Benefit of Fully Assoc Cache
  - No Conflict Misses (since data can go anywhere)

- Drawbacks of Fully Assoc Cache
  - Need hardware comparator for every single block: if we have a 64KB of data in cache with 4B blocks, we need 16K comparators: infeasible
Final Type of Cache Miss

• **3rd C: Capacity Misses**
  - miss that occurs because the cache has a limited size
  - miss that would not occur if we increase the size of the cache
  - sketchy definition, so just get the general idea

• This is the primary type of miss for Fully Associative caches.

N-Way Set Associative Cache (1/3)

• Memory address fields:
  - Tag: same as before
  - Offset: same as before
  - Index: points us to the correct “row” (called a set in this case)

• So what’s the difference?
  - each set contains multiple blocks
  - once we’ve found correct set, must compare with all tags in that set to find our data

N-Way Set Associative Cache (2/3)

• Basic Idea
  - cache is direct-mapped w/respect to sets
  - each set is fully associative with N blocks in it

• Given memory address:
  - Find correct set using Index value.
  - Compare Tag with all Tag values in the determined set.
  - If a match occurs, hit!, otherwise a miss.
  - Finally, use the offset field as usual to find the desired data within the block.

N-Way Set Associative Cache (3/3)

• What’s so great about this?
  - even a 2-way set assoc cache avoids a lot of conflict misses
  - hardware cost isn’t that bad: only need N comparators

• In fact, for a cache with M blocks,
  - it’s Direct-Mapped if it’s 1-way set assoc
  - it’s Fully Assoc if it’s M-way set assoc
  - so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit

- [Diagram of 4-way set associative cache circuit]
Block Replacement Policy

- **Direct-Mapped Cache**
  - index completely specifies position which position a block can go in on a miss

- **N-Way Set Assoc**
  - index specifies a set, but block can occupy any position within the set on a miss

- **Fully Associative**
  - block can be written into any position

**Question:** If we have the choice, where should we write an incoming block?

- If there are any locations with valid bit off (empty), then usually write the new block into the first one.
- If all possible locations already have a valid block, we must pick a replacement policy: rule by which we determine which block gets “cached out” on a miss.

Block Replacement Policy: **LRU**

- **LRU (Least Recently Used)**
  - Idea: cache out block which has been accessed (read or write) least recently
  - Pro: temporal locality ⇒ recent past use implies likely future use: in fact, this is a very effective policy
  - Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Block Replacement Example

- We have a 2-way set associative cache with a four word total capacity and one word blocks. We perform the following word accesses (ignore bytes for this problem):
  - 0, 2, 0, 1, 4, 0, 2, 3, 5, 4
- How many hits and how many misses will there be for the LRU block replacement policy?

**Big Idea**

- How to choose between associativity, block size, replacement & write policy?
- Design against a performance model
  - Minimize: Average Memory Access Time = Hit Time + Miss Penalty x Miss Rate
  - Influenced by technology & program behavior, architecture, cache purpose
- Create the illusion of a memory that is large, cheap, and fast - on average
- How can we improve miss penalty?

Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM ⇒ 200 processor clock cycles!

Solution: another cache between memory and the processor cache: **Second Level (L2) Cache**
Peer Instruction

1. A 2-way set-associative cache can be outperformed by a direct-mapped cache.
2. Larger block size ⇒ lower miss rate

And in Conclusion...

• We've discussed memory caching in detail. Caching in general shows up over and over in computer systems
• Filesystem cache, Web page cache, Game databases / tablebases, Software memoization, and many more!
• Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.
• Cache design choices:
  • Size of cache: speed v. capacity
  • Block size (i.e., cache aspect ratio)
  • Write Policy (Write through vs. write back)
  • Associativity choice of N (direct-mapped vs. set vs. fully associative)
  • Block replacement policy
  • 2nd level cache?
  • 3rd level cache?
• Use performance model to pick between choices, depending on programs, technology, budget, ...

Bonus slides

• These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
• The slides will appear in the order they would have in the normal presentation

Example

• Assume
  • Hit Time = 1 cycle
  • Miss rate = 5%
  • Miss penalty = 20 cycles
  • Calculate AMAT...
• Avg mem access time
  = 1 + 0.05 x 20
  = 1 + 1 cycles
  = 2 cycles

Ways to reduce miss rate

• Larger cache
  • limited by cost and technology
  • hit time of first level cache < cycle time (bigger caches are slower)
• More places in the cache to put each block of memory – associativity
  • fully-associative
    • any block any line
  • N-way set associated
    • N places for each block
    • direct map: N=1

Analyzing Multi-level cache hierarchy

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty =
L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)
Typical Scale

- **L1**
  - size: tens of KB
  - hit time: complete in one clock cycle
  - miss rates: 1-5%

- **L2**
  - size: hundreds of KB
  - hit time: few clock cycles
  - miss rates: 10-20%

- L2 miss rate is fraction of L1 misses that also miss in L2
  - why so high?

Example: with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 200 cycles
  - L1 miss penalty = 5 + 0.15 * 200 = 35
  - Avg mem access time = 1 + 0.05 x 35 = 2.75 cycles

Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
  - Avg mem access time = 1 + 0.05 x 200 = 11 cycles

- 4x faster with L2 cache! (2.75 vs. 11)

An actual CPU – Early PowerPC

- **Cache**
  - 32 KB Instructions and 32 KB Data L1 caches
  - External L2 Cache interface with integrated controller and cache tags, supports up to 1 MByte external L2 cache
  - Dual Memory Management Units (MMU) with Translation Lookaside Buffers (TLB)

- **Pipelining**
  - Superscalar (3 inst/cycle)
  - 6 execution units (2 integer and 1 double precision IEEE floating point)

An Actual CPU – Pentium M

- [Image of Pentium M processor]

- [Image of Pentium M processor map]