Caches 2

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

P & H Chapter 5.2 (writes), 5.3, 5.5
<table>
<thead>
<tr>
<th>Direct Mapped</th>
<th>Fully Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Smaller</td>
<td>Tag Size</td>
</tr>
<tr>
<td>+ Less</td>
<td>SRAM Overhead</td>
</tr>
<tr>
<td>+ Less</td>
<td>Controller Logic</td>
</tr>
<tr>
<td>+ Faster</td>
<td>Speed</td>
</tr>
<tr>
<td>+ Less</td>
<td>Price</td>
</tr>
<tr>
<td>+ Very</td>
<td>Scalability</td>
</tr>
<tr>
<td>– Lots</td>
<td># of conflict misses</td>
</tr>
<tr>
<td>– Low</td>
<td>Hit rate</td>
</tr>
<tr>
<td>– Common</td>
<td>Pathological Cases?</td>
</tr>
<tr>
<td></td>
<td>Larger –</td>
</tr>
<tr>
<td></td>
<td>More –</td>
</tr>
<tr>
<td></td>
<td>More –</td>
</tr>
<tr>
<td></td>
<td>Slower –</td>
</tr>
<tr>
<td></td>
<td>More –</td>
</tr>
<tr>
<td></td>
<td>Not Very –</td>
</tr>
<tr>
<td></td>
<td>Zero +</td>
</tr>
<tr>
<td></td>
<td>High +</td>
</tr>
<tr>
<td></td>
<td>?</td>
</tr>
</tbody>
</table>
Set Associative Caches
Set Associative Cache

- Each block number mapped to a single cache line set index
- Within the set, block can go in any line

<table>
<thead>
<tr>
<th>set 0</th>
<th>line 0</th>
<th></th>
<th>line 1</th>
<th></th>
<th>line 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set 1</td>
<td>line 3</td>
<td></td>
<td>line 4</td>
<td></td>
<td>line 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compromise

0x000000
0x000004
0x000008
0x00000c
0x000010
0x000014
0x000018
0x00001c
0x000020
0x000024
0x000028
0x00002c
0x000030
0x000034
0x000038
0x00003c
0x000040
0x000044
0x000048
0x00004c
Set Associative Cache

Like direct mapped cache

• Only need to check a few lines for each access... so: fast, scalable, low overhead

Like a fully associative cache

• Several places each block can go... so: fewer conflict misses, higher hit rate
Using **byte addresses** in this example! Addr Bus = 5 bits

**Processor**
- `lb $1 ← M[ 1 ]`
- `lb $2 ← M[ 13 ]`
- `lb $3 ← M[ 0 ]`
- `lb $3 ← M[ 6 ]`
- `lb $2 ← M[ 5 ]`
- `lb $2 ← M[ 6 ]`
- `lb $2 ← M[ 10 ]`
- `lb $2 ← M[ 12 ]`

**2-Way Set Associative Cache**

\[ A = \]

\[
\begin{array}{c|c|c}
V & \text{tag} & \text{data} \\
\hline
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
& & \\
\end{array}
\]

**Memory**

- 0: 101
- 1: 103
- 2: 107
- 3: 109
- 4: 113
- 5: 127
- 6: 131
- 7: 137
- 8: 139
- 9: 149
- 10: 151
- 11: 157
- 12: 163
- 13: 167
- 14: 173
- 15: 179
- 16: 181

**Hits:** **Misses:**
### A Pathological Case

<table>
<thead>
<tr>
<th>Processor</th>
<th>Direct Mapped</th>
<th>2-Way Set Associative</th>
<th>Fully Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb $1 \leftarrow M[1]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $2 \leftarrow M[8]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $3 \leftarrow M[1]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $3 \leftarrow M[8]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $2 \leftarrow M[1]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $2 \leftarrow M[16]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $2 \leftarrow M[1]$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lb $2 \leftarrow M[8]$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</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>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
</tbody>
</table>
To Do:

- Evicting cache lines
- Picking cache parameters
- Writing using the cache
Cache Parameters
direct mapped, 2-way, 8-way, fully associative
Q: Which line should we evict to make room?
For direct-mapped?
A: no choice, must evict the indexed line
For associative caches?
A: FIFO: oldest line (timestamp per line)
A: LRU: least recently used (ts per line)
A: LFU: (need a counter per line)
A: MRU: most recently used (?) (ts per line)
A: round-robin (need a finger per set)
A: random (free!)
A: Belady’s: optimal (need time travel)
Need to determine parameters:

- Block size (aka line size)
- Number of ways of set-associativity \((1, N, \infty)\)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies
- Write policy
> dmidecode -t cache
Cache Information
  Configuration: Enabled, Not Socketed, Level 1
  Operational Mode: Write Back
  Installed Size: 128 KB
  Error Correction Type: None
Cache Information
  Configuration: Enabled, Not Socketed, Level 2
  Operational Mode: Varies With Memory Address
  Installed Size: 6144 KB
  Error Correction Type: Single-bit ECC
> cd /sys/devices/system/cpu/cpu0; grep cache/*/ *
cache/index0/level:1
cache/index0/type: Data
cache/index0/ways_of_associativity: 8
cache/index0/number_of_sets: 64
cache/index0/coherency_line_size: 64
cache/index0/size: 32K
cache/index1/level: 1
cache/index1/type: Instruction
cache/index1/ways_of_associativity: 8
cache/index1/number_of_sets: 64
cache/index1/coherency_line_size: 64
cache/index1/size: 32K
cache/index2/level: 2
cache/index2/type: Unified
cache/index2/shared_cpu_list: 0-1
cache/index2/ways_of_associativity: 24
cache/index2/number_of_sets: 4096
cache/index2/coherency_line_size: 64
cache/index2/size: 6144K

Dual-core 3.16GHz Intel (purchased in 2009)
Dual 32K L1 Instruction caches
• 8-way set associative
• 64 sets
• 64 byte line size

Dual 32K L1 Data caches
• Same as above

Single 6M L2 Unified cache
• 24-way set associative (~!!!)
• 4096 sets
• 64 byte line size

4GB Main memory
1TB Disk

Dual-core 3.16GHz Intel
(purchased in 2009)
Q: How to decide block size?
A: Try it and see
But: depends on cache size, workload, associativity, ...
For a given total cache size, larger block sizes mean....

- fewer lines
- so fewer tags (and smaller tags for associative caches)
- so less overhead
- and fewer cold misses (within-block “prefetching”)

But

- fewer blocks available (for scattered accesses!)
- so more conflicts
- and miss penalty (time to fetch block) is larger
Writing with Caches
Q: How to write data?

If data is already in the cache...

No-Write
- writes invalidate the cache and go directly to memory

Write-Through
- writes go to main memory and cache

Write-Back
- CPU writes only to cache
- cache writes to main memory later (when block is evicted)
Q: How to write data?

If data is not in the cache...

Write-Allocate
- allocate a cache line for new data (and maybe write-through)

No-Write-Allocate
- ignore cache, just go to main memory
Using **byte addresses** in this example! Addr Bus = 5 bits

<table>
<thead>
<tr>
<th>Processor</th>
<th>Direct Mapped Cache + Write-through + Write-allocate</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>lb $1 \leftarrow M[1]$</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>lb $2 \leftarrow M[7]$</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>sb $2 \rightarrow M[0]$</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>sb $1 \rightarrow M[5]$</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>lb $2 \leftarrow M[9]$</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>sb $1 \rightarrow M[5]$</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>sb $1 \rightarrow M[0]$</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>$1$</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>$2$</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>$3$</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>$4$</td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>

**V**  | **tag**  | **data**  |

| Hits: | Misses: |

<table>
<thead>
<tr>
<th>0</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>103</td>
</tr>
<tr>
<td>2</td>
<td>107</td>
</tr>
<tr>
<td>3</td>
<td>109</td>
</tr>
<tr>
<td>4</td>
<td>113</td>
</tr>
<tr>
<td>5</td>
<td>127</td>
</tr>
<tr>
<td>6</td>
<td>131</td>
</tr>
<tr>
<td>7</td>
<td>137</td>
</tr>
<tr>
<td>8</td>
<td>139</td>
</tr>
<tr>
<td>9</td>
<td>149</td>
</tr>
<tr>
<td>10</td>
<td>151</td>
</tr>
<tr>
<td>11</td>
<td>157</td>
</tr>
<tr>
<td>12</td>
<td>163</td>
</tr>
<tr>
<td>13</td>
<td>167</td>
</tr>
<tr>
<td>14</td>
<td>173</td>
</tr>
<tr>
<td>15</td>
<td>179</td>
</tr>
<tr>
<td>16</td>
<td>181</td>
</tr>
</tbody>
</table>
Write-through performance

Each miss (read or write) reads a block from mem
  • 5 misses $\rightarrow$ 10 mem reads
Each store writes an item to mem
  • 4 mem writes
Using byte addresses in this example! Addr Bus = 5 bits

Processor

lb $1 ← M[1]
lb $2 ← M[7]
sb $2 → M[0]
sb $1 → M[5]
lb $2 ← M[9]
sb $1 → M[5]
sb $1 → M[0]

$1
$2
$3
$4

Direct Mapped Cache
+ Write-back
+ Write-allocate

Memory

0 101
1 103
2 107
3 109
4 113
5 127
6 131
7 137
8 139
9 149
10 151
11 157
12 163
13 167
14 173
15 179
16 181

V D tag data

Hits: Misses:
Write-back performance

Each miss (read or write) reads a block from mem

• 5 misses $\rightarrow$ 10 mem reads

Some evictions write a block to mem

• 1 dirty eviction $\rightarrow$ 2 mem writes
• (+ 2 dirty evictions later $\rightarrow$ +4 mem writes)
<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>Tag</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>...</th>
<th>Byte N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

V = 1 means the line has valid data

D = 1 means the bytes are newer than main memory

When allocating line:

- Set V = 1, D = 0, fill in Tag and Data

When writing line:

- Set D = 1

When evicting line:

- If D = 0: just set V = 0
- If D = 1: write-back Data, then set D = 0, V = 0
Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

```c
for (i=1; i<n; i++)
    A[0] += A[i];

for (i=0; i<n; i++)
    B[i] = A[i]
```
Q: Hit time: write-through vs. write-back?
A: Write-through slower on writes.

Q: Miss penalty: write-through vs. write-back?
A: Write-back slower on evictions.
Q: Writes to main memory are slow!
A: Use a write-back buffer
   • A small queue holding dirty lines
   • Add to end upon eviction
   • Remove from front upon completion

Q: What does it help?
A: short bursts of writes (but not sustained writes)
A: fast eviction reduces miss penalty
Write-through is slower
  • But simpler (memory always consistent)

Write-back is almost always faster
  • But what about multiple cores sharing memory?
// H = 12, W = 10
int A[H][W];

for(x=0; x < W; x++)
    for(y=0; y < H; y++)
        sum += A[y][x];

Every access is a cache miss!
(unless *entire* matrix can fit in cache)
// H = 12, W = 10
int A[H][W];

for(y=0; y < H; y++)
    for(x=0; x < W; x++)
        sum += A[y][x];

Block size = 4  \rightarrow  75\% hit rate
Block size = 8  \rightarrow  87.5\% hit rate
Block size = 16  \rightarrow  93.75\% hit rate
And you can easily prefetch to warm the cache.
Memory performance matters!
- often more than CPU performance
- ... because it is the bottleneck, and not improving much
- ... because most programs move a LOT of data

Design space is huge
- Gambling against program behavior
- Cuts across all layers:
  users → programs → os → hardware

Multi-core / Multi-Processor is complicated
- Inconsistent views of memory
- Need to "snoop" in each other’s caches
- Extremely complex protocols, very hard to get right