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></th>
<th>Direct Mapped</th>
<th>Fully Associative</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Smaller</td>
<td>Larger −</td>
</tr>
<tr>
<td></td>
<td>+ Less</td>
<td>More −</td>
</tr>
<tr>
<td></td>
<td>+ Less</td>
<td>More −</td>
</tr>
<tr>
<td></td>
<td>+ Faster</td>
<td>Slower −</td>
</tr>
<tr>
<td></td>
<td>+ Very</td>
<td>More −</td>
</tr>
<tr>
<td></td>
<td>− Lots</td>
<td>Not Very −</td>
</tr>
<tr>
<td></td>
<td>− Low</td>
<td>Zero +</td>
</tr>
<tr>
<td></td>
<td>− Common</td>
<td>High +</td>
</tr>
<tr>
<td>Tag Size</td>
<td>SRAM Overhead</td>
<td></td>
</tr>
<tr>
<td>Controller Logic</td>
<td>Speed</td>
<td></td>
</tr>
<tr>
<td>Price</td>
<td>Scalability</td>
<td></td>
</tr>
<tr>
<td># of conflict misses</td>
<td>Hit rate</td>
<td></td>
</tr>
<tr>
<td>Pathological Cases?</td>
<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
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 ]`

### Memory

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

### 2-Way Set Associative Cache

- `V` (valid)
- `tag`
- `data`

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>000</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>001</td>
<td>131</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

#### Processor
- lb $1 \leftarrow M[1]
- lb $2 \leftarrow M[8]
- lb $3 \leftarrow M[1]
- lb $3 \leftarrow M[8]
- lb $2 \leftarrow M[1]
- lb $2 \leftarrow M[16]
- lb $2 \leftarrow M[1]
- lb $2 \leftarrow M[8]

#### Memory

<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>

#### Direct Mapped

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

#### 2-Way Set Associative

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

#### Fully Associative

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
</tr>
</tbody>
</table>

**8 Misses**
To Do:

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

direct mapped, 2-way, 8-way, fully associative

Cache size →

Miss rate →

Worse

Conflict

Cold

Capacity

Conflict

A Cold

$
Need to determine parameters:

- Cache size
- Block size (aka line size)
- Number of ways of set-associativity (1, N, ∞)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy
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

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, ...

Experimental approach!
Experimental Results
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 also...

- fewer blocks available (for scattered accesses!)
- so more conflicts
- and larger miss penalty (time to fetch block)
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

### 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]`

### Direct Mapped Cache
+ Write-through
+ Write-allocate

### Memory

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>101</td>
</tr>
<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>

### Hits: Misses:
- Hits: [113, 131, 137, 163, 179]
- Misses: [127, 137, 163, 181]
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
Evictions don’t need to write to mem
  • no need for dirty bit
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 ]`

**Direct Mapped Cache**
- + Write-back
- + Write-allocate

**Memory**

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

**V D tag data**

<table>
<thead>
<tr>
<th>Hits:</th>
<th>Misses:</th>
</tr>
</thead>
</table>

A Simple 2-Way Set Associative Cache
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)
• need a dirty bit
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

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

```java
for (i=0; i<n; i++)
    B[i] = A[i]
```
Performance: Write-back versus Write-through

Assume: large associative cache, 16-byte lines

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
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

• write-back buffer hides large eviction cost
• But what about multiple cores with separate caches but sharing memory?

Write-back requires a cache coherency protocol

• Inconsistent views of memory
• Need to “snoop” in each other’s caches
• Extremely complex protocols, very hard to get right
Q: Multiple readers and writers?
A: Potentially inconsistent views of memory

Cache coherency protocol

- May need to **snoop** on other CPU’s cache activity
- **Invalidate** cache line when other CPU writes
- **Flush** write-back caches before other CPU reads
- Or the reverse: Before writing/reading...
- Extremely complex protocols, very hard to get right
Cache Conscious Programming
// 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  →  75% hit rate
Block size = 8  →  87.5% hit rate
Block size = 16  →  93.75% hit rate
And you can easily prefetch to warm the cache.
Caching assumptions

• small working set: 90/10 rule
• can predict future: spatial & temporal locality

Benefits

• (big & fast) built from (big & slow) + (small & fast)

Tradeoffs:

associativity, line size, hit cost, miss penalty, hit rate
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 $\rightarrow$ programs $\rightarrow$ os $\rightarrow$ hardware

Multi-core / Multi-Processor is complicated

- Inconsistent views of memory
- Extremely complex protocols, very hard to get right