Caches 3

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

See P&H Chapter: 5.1-5.4, 5.8, 5.10, 5.15; Also, 5.13 & 5.17
Overview

Writing to caches: policies, performance

Cache tradeoffs and performance
What about Stores?

Where should you write the result of a store?

- If that memory location is in the cache?
  - Send it to the cache
  - Should we also send it to memory right away? (write-through policy)
  - Wait until we evict the block (write-back policy)

- If it is not in the cache?
  - Allocate the line (put it in the cache)? (write allocate policy)
  - Write it directly to memory without allocation? (no write allocate policy)
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
Next Goal

How does a write-through cache work?
Assume write-allocate
Handling Stores (Write-Through)

Using **byte addresses** in this example! Addr Bus = 5 bits

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume write-allocate policy</td>
<td>Fully Associative Cache</td>
<td></td>
</tr>
<tr>
<td>LB $1 ← M[ 1 ]</td>
<td>2 cache lines</td>
<td>0</td>
</tr>
<tr>
<td>LB $2 ← M[ 7 ]</td>
<td>2 word block</td>
<td>1</td>
</tr>
<tr>
<td>SB $2 → M[ 0 ]</td>
<td>4 bit tag field</td>
<td>2</td>
</tr>
<tr>
<td>SB $1 → M[ 5 ]</td>
<td>1 bit block offset field</td>
<td>3</td>
</tr>
<tr>
<td>SB $1 → M[ 10 ]</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>$0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>$1</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>$2</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>$3</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

Misses: 0

Hits: 0

Using byte addresses in this example! Addr Bus = 5 bits
Write-Through (REF 1)

Processor:
- LB $1 ← M[ 1 ]
- LB $2 ← M[ 7 ]
- SB $2 → M[ 0 ]
- SB $1 → M[ 5 ]
- LB $2 ← M[ 10 ]
- SB $1 → M[ 5 ]
- SB $1 → M[ 10 ]

Cache:
- Misses: 0
- Hits: 0

Memory:
- Misses: 0
- Hits: 0

CacheProcessor

V tag data

0
0

0
0

$0
$1
$2
$3

Memory:
- 0: 78
- 1: 29
- 2: 120
- 3: 123
- 4: 71
- 5: 150
- 6: 162
- 7: 173
- 8: 18
- 9: 21
- 10: 33
- 11: 28
- 12: 19
- 13: 200
- 14: 210
- 15: 225
How Many Memory References?

Write-through performance

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

Each store writes an item to mem

Evictions don’t need to write to mem
Summary: Write Through

Write-through policy with write allocate

Cache miss: read entire block from memory

Write: write only updated item to memory

Eviction: no need to write to memory
Write-Through vs. Write-Back

Can we also design the cache **NOT** to write all stores immediately to memory?

- Keep the most current copy in cache, and update memory when that data is evicted (*write-back policy*)
- Do we need to write-back all evicted lines?
  - No, only blocks that have been stored into (written)
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
Write-back Example

Example: How does a write-back cache work?
Assume write-allocate
Handling Stores (Write-Back)

Using **byte addresses** in this example! Addr Bus = 5 bits

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cache</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Assume write-allocate policy</strong></td>
<td><strong>Fully Associative Cache</strong>&lt;br&gt;2 cache lines&lt;br&gt;2 word block&lt;br&gt;3 bit tag field&lt;br&gt;1 bit block offset field</td>
<td></td>
</tr>
<tr>
<td>LB $1 \rightarrow M[ 1 ]</td>
<td>V d tag data</td>
<td></td>
</tr>
<tr>
<td>LB $2 \rightarrow M[ 7 ]</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>SB $2 \rightarrow M[ 0 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB $1 \rightarrow M[ 5 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LB $2 \rightarrow M[ 10 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB $1 \rightarrow M[ 5 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB $1 \rightarrow M[ 10 ]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0</td>
<td></td>
<td>150</td>
</tr>
<tr>
<td>$1</td>
<td></td>
<td>162</td>
</tr>
<tr>
<td>$2</td>
<td></td>
<td>173</td>
</tr>
<tr>
<td>$3</td>
<td></td>
<td>18</td>
</tr>
</tbody>
</table>

Misses: 0

Hits: 0

Using **byte addresses** in this example! Addr Bus = 5 bits

Assume write-allocate policy

$
\begin{align*}
&LB \; \$1 \rightarrow M[1] \\
&LB \; \$2 \rightarrow M[7] \\
&SB \; \$2 \rightarrow M[0] \\
&SB \; \$1 \rightarrow M[5] \\
&LB \; \$2 \rightarrow M[10] \\
&SB \; \$1 \rightarrow M[5] \\
&SB \; \$1 \rightarrow M[10] \\
\end{align*}
$

$\begin{array}{|c|}
\hline
\$0 \\
\$1 \\
\$2 \\
\$3 \\
\hline
\end{array}$

$\begin{array}{|c|}
\hline
V \\
d \\
tag \\
data \\
\hline
0 \\
\hline
\end{array}$

$\begin{array}{|c|}
\hline
0 \\
\hline
\end{array}$

Memory:

<table>
<thead>
<tr>
<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>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>29</td>
<td>120</td>
<td>123</td>
<td>71</td>
<td>150</td>
<td>162</td>
<td>173</td>
<td>18</td>
<td>21</td>
<td>33</td>
<td>28</td>
<td>19</td>
<td>200</td>
<td>210</td>
<td>225</td>
</tr>
</tbody>
</table>
Write-Back (REF 1)

Processor

Cache

Memory

 LB $1 \leftarrow M[1]$
 LB $2 \leftarrow M[7]$
 SB $2 \rightarrow M[0]$
 SB $1 \rightarrow M[5]$
 LB $2 \leftarrow M[10]$
 SB $1 \rightarrow M[5]$
 SB $1 \rightarrow M[10]$

$0$

$1$

$2$

$3$

$0$

$1$

$2$

$3$

$0$

$1$

$2$

$3$

Misses: 0

Hits: 0

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

78

29

120

123

71

150

162

173

18

21

33

28

19

200

210

225
How Many Memory References?

Write-back performance

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

*Some* evictions write a block to mem
So is write back just better?

What are other performance tradeoffs between write-through and write-back?

How can we further reduce penalty for cost of writes to memory?
Performance: An Example

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];
```
Performance Tradeoffs

Q: Hit time: write-through vs. write-back?

Q: Miss penalty: write-through vs. write-back?
Write Buffering

Q: Writes to main memory are slow!

Q: When does it help?
Write-through vs. Write-back

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
Summary: Write Through

Write-through policy with write allocate
- Cache miss: read entire block from memory
- Write: write only updated item to memory
- Eviction: no need to write to memory
- **Slower, but cleaner**

Write-back policy with write allocate
- Cache miss: read entire block from memory
  - But may need to write dirty cacheline first
- Write: nothing to memory
- Eviction: have to write to memory, entire cacheline because don’t know what is dirty (only 1 dirty bit)
- **Faster, but complicated with multicore**
Next Goal

Performance: What is the average memory access time (AMAT) for a cache?

$$AMAT = \%hit \times \text{hit time} + \% \text{miss} \times \text{miss time}$$
Cache Performance Example

Average Memory Access Time (AMAT)

Cache Performance (very simplified):

L1 (SRAM): 512 x 64 byte cache lines, direct mapped
  Data cost: 3 cycle per word access
  Lookup cost: 2 cycle

Mem (DRAM): 4GB
  Data cost: 50 cycle for first word, plus 3 cycles per subsequent word

16 words (i.e. 64 / 4 = 16)
Average Memory Access Time (AMAT)

Cache Performance (very simplified):

**L1 (SRAM):** 512 x 64 byte cache lines, direct mapped
  - Data cost: 3 cycle per word access
  - Lookup cost: 2 cycle
  - 16 words (i.e. 64 / 4 = 16)

**Mem (DRAM):** 4GB
  - Data cost: 50 cycle for first word, plus 3 cycles per subsequent word
Multi Level Caching

Cache Performance (very simplified):

**L1 (SRAM):** 512 x 64 byte cache lines, direct mapped
   Hit time: 5 cycles

L2 cache: bigger
   Hit time = 20 cycles

**Mem (DRAM):** 4GB
   Hit rate: 90% in L1, 90% in L2

Often: L1 fast and direct mapped, L2 bigger and higher associativity
Performance Summary

Average memory access time (AMAT) depends on cache architecture and size
access time for hit,
miss penalty, miss rate

Cache design a very complex problem:

• 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
// H = 12, W = 10
int A[H][W];

for (x=0; x < W; x++)
    for (y=0; y < H; y++)
        sum += A[y][x];
// H = 12, W = 10
int A[H][W];

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

Dual-core 3.16GHz Intel (purchased in 2011)
A Real Example

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)
Figure 1. Schematic diagram of a Sandy Bridge processor.
Summary

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