Cache Issues

Why the book? Lecture notes are faster!

Quiz #3 Friday!

Basic Cache Algorithm

ON REFERENCE TO Mem[X]:
Look for X among cache tags...

HIT: X = TAG[i], for some cache line i
  • READ: return DATA(i)
  • WRITE: change DATA(i); Start Write to Mem(X)

MISS: X not found in TAG of any cache line
  • REPLACEMENT SELECTION:
    • Select some line k to hold Mem[X] (Allocation)
  • READ: Read Mem[X]
    Set TAG(k)=X, DATA(K)=Mem[X]
  • WRITE: Start Write to Mem(X)
    Set TAG(k)=X, DATA(K)= new Mem[X]

Cache Design Issues

Associativity – a basic tradeoff between
  • Parallel Searching (expensive) vs
  • Constraints on which addresses can be stored where

Replacement Strategy:
  • OK, we’ve missed. Gotta add this new address/value pair to the cache. What do we kick out?
    • Least Recently Used: discard the one we haven’t used the longest.
    • Plausible alternatives, (e.g. random replacement.

Block Size:
  • Amortizing cost of tag over multiple words of data

Write Strategy:
  • When do we write cache contents to main memory?

Associativity

Fully Associative
  - expensive!
  - flexible: any address can be cached in any line

Direct Mapped
  - cheap (ordinary SRAM)
  - contention: addresses compete for cache lines

... or NONE!
Direct-Mapped Cache Contention

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Line</th>
<th>Hit/Hit</th>
<th>Miss/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>HIT</td>
<td></td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>HIT</td>
<td></td>
</tr>
</tbody>
</table>

Works GREAT here...

Assume 1024-line direct-mapped cache, 1 word/line. Consider tight loop, at steady state:
(assume WORD, not BYTE, addressing)

Loop A:
Pgms at 1024,
data at 37:

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache Line</th>
<th>Hit/Hit</th>
<th>Miss/Miss</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>1025</td>
<td>1</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>2049</td>
<td>1</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>1026</td>
<td>2</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>2050</td>
<td>2</td>
<td>MISS</td>
<td>MISS</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>MISS</td>
<td>MISS</td>
</tr>
</tbody>
</table>

... but not here!

We need some associativity. But not full associativity...

Cost vs Contention

two observations...

1. Probability of collision diminishes with cache size...
   ... so let's build HUGE direct-mapped caches, using cheap SRAM!

2. Contention mostly occurs between independent "hot spots" --
   • Instruction fetches vs stack frame vs data structures, etc
   • Ability to simultaneously cache a few (2? 4? 8?)
     hot spots eliminates most collisions
   ... so let's build caches that allow each location to be
   stored in some restricted set of cache lines,
   rather than in exactly one (direct mapped) or
   every line (fully associative).

Insight: an N-way set-associative cache affords modest parallelism
   • parallel lookup (associativity): restricted to small set of N lines
   • modest parallelism deals with most contention at modest cost
   • can implement using N direct-mapped caches, running in parallel

Fully-associative vs. Direct-mapped

Fully-associative N-line cache:
- N tag comparators, registers used
  for tag/data storage ($$)
- Location A might be cached in any one
  of the N cache lines; no restrictions!
- Replacement strategy (e.g., LRU)
  used to pick which line to use when
  loading new word(s) into cache
- PROBLEM: Cost!

Direct-mapped N-line cache:
- 1 tag comparator, SRAM used for
  tag/data storage ($)
- Location A is cached in a specific line
  of the cache determined by its
  address; address "collisions" possible
- Replacement strategy not needed:
  each word can only be cached in one
  specific cache line
- PROBLEM: Contention!

N-way Set-Associative Cache

Simultaneously address line in each subcache constitutes a set

MEM DATA

DATA TO CPU

HIT

INCOMING ADDRESS

N direct-mapped caches, each with 2^t lines

k

t
**Associativity**

the birds-eye view

Can place caches in 2D space:

- Total lines = \# Sets \* Set Size
- \# Sets = 1: Fully Associative
- Set Size = 1: Direct Mapped
- Set Size = N: N-way Set Associative

**Replacement Strategy**

LRU (Least-recently used)

- keeps most-recently used locations in cache
- need to keep ordered list of N items \(\rightarrow\) \(N!\) orderings
  \(\rightarrow\) \(O(\log_2 N)\) "LRU bits" + complex logic

FIFO/LRR (first-in, first-out/least-recently replaced)

- cheap alternative: replace oldest item (dated by access time)
- within each set: keep one counter that points to victim line

Random (select replacement line using random, uniform distribution)

- no "pathological" reference streams causing worst-case results
- use pseudo-random generator to get reproducible behavior;
  use real randomness to prevent reverse engineering!

**Cache Benchmarking**

Suppose this loop is entered with R3=4000:

<table>
<thead>
<tr>
<th>ADR</th>
<th>Instruction</th>
<th>I</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>LD (R3, 0, R0)</td>
<td>400</td>
<td>4000+...</td>
</tr>
<tr>
<td>404</td>
<td>ADDC (R3, 4, R3)</td>
<td>404</td>
<td></td>
</tr>
<tr>
<td>408</td>
<td>BNE (R0, 400)</td>
<td>408</td>
<td></td>
</tr>
</tbody>
</table>

**GOAL:** Given some cache design, simulate (by hand or machine) execution well enough to determine hit ratio.

1. Observe that the sequence of memory locations referenced is 400, 4000, 404, 408, 400, 4004, ...

We can use this simpler reference string, rather than the program, to simulate cache behavior.

2. We can make our life easier in many cases by converting to word addresses: 100, 1000, 101, 102, 100, 1001,...

\(\text{Word Addr} = (\text{Byte Addr})/4\)
Cache Simulation

4-line Fully-associative/LRU

<table>
<thead>
<tr>
<th>Addr</th>
<th>Line#</th>
<th>Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1002</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>H</td>
</tr>
<tr>
<td>1003</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
</tbody>
</table>

Compulsory Misses

1/4 miss

7/16 miss

4-line Direct-mapped

<table>
<thead>
<tr>
<th>Addr</th>
<th>Line#</th>
<th>Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1002</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1003</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
</tbody>
</table>

Assocativity: Full vs 2-way

8-line Fully-associative, LRU

<table>
<thead>
<tr>
<th>Addr</th>
<th>Line#</th>
<th>Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>3</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1002</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1003</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
</tbody>
</table>

7/16 miss

1/4 miss

4-line Direct-mapped

<table>
<thead>
<tr>
<th>Addr</th>
<th>Line#</th>
<th>Miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1002</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>M</td>
</tr>
<tr>
<td>1003</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>M</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>M</td>
</tr>
</tbody>
</table>

Devil’s Advocacy Games

Your company uses the cheaper FIFO cache, the competition uses LRU. Can you devise a benchmark to make your cache look better?

Assume 0x100 sets, 2-way…

A carefully-designed benchmark can make either look better…

Pessimal case: next addr referenced is the one just replaced!

Random replacement makes this game harder…

1-way (direct-mapped)

2-way

4-way

8-way

fully assoc.

Associativity vs. miss rate

Miss rate (%)

H&P: Figure 5.9

Associativity

• 8-way is (almost) as effective as fully-associative
• rule of thumb: N-line direct-mapped == N/2-line 2-way set assoc.

Miss rate (%)

12
10
8
6
4
2
0
1k 2k 4k 8k 16k 32k 64k 128k

Cache size (bytes)

Assosciativity

1-way

2-way

4-way

8-way

fully assoc.
**Increasing Block Size**

More Data/Tag

Enlarge each line in cache:

28 bits

TAG

D0

D1

D2

D3

4 x 32 = 128 bits

ADDR

[3:2]

[31:4]

32

DATA

HIT

Overhead < ¼ bit of Tag per bit of data

- blocks of $2^b$ words, on $2^b$ word boundaries
- always read/write $2^b$ word block from/to memory
- locality: access on word in block, others likely
- cost: some fetches of unaccessed words

BIG WIN if there is a wide path to memory

---

**4-word block, DM Cache**

4-bit index

0x12 M[0x1230]  M[0x1234]  M[0x1238]  M[0x123C]

0x12 M[0x1240]  M[0x1244]  M[0x1248]  M[0x124C]

Use ordinary (fast) static RAM for tag and data storage

Only one comparator for entire cache!

---

**Handling of WRITES**

Observation: Most (90+) of memory accesses are READs. How should we handle writes? Issues:

- **Write-through**: CPU writes are cached, but also written to main memory (stalling the CPU until write is completed). Memory always holds "the truth".
- **Write-behind**: CPU writes are cached; writes to main memory may be buffered, perhaps pipelined. CPU keeps executing while writes are completed (in order) in the background.
- **Write-back**: CPU writes are cached, but not immediately written to main memory. Memory contents can be "stale".

Our cache thus far uses write-through.

Can we improve write performance?
**Write-through**

ON REFERENCE TO Mem[X]: Look for X among tags...

**HIT:** X == TAG(i), for some cache line i

- **READ:** return DATA[i]
- **WRITE:** change DATA[i]; Start Write to Mem[X]

**MISS:** X not found in TAG of any cache line

- **REPLACEMENT SELECTION:** Select some line k to hold Mem[X]
- **READ:** Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X]
- **WRITE:** Start Write to Mem[X]
  - Set TAG[k] = X, DATA[k] = new Mem[X]

---

**Write-back**

ON REFERENCE TO Mem[X]: Look for X among tags...

**HIT:** X = TAG(i), for some cache line i

- **READ:** return DATA[i]
- **WRITE:** change DATA[i]; Start Write to Mem[X]

**MISS:** X not found in TAG of any cache line

- **REPLACEMENT SELECTION:**
  - Select some line k to hold Mem[X]
  - Write Back: Write Data(k) to Mem[Tag[k]]
- **READ:** Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X]
- **WRITE:** Start Write to Mem[X]
  - Set TAG[k] = X, DATA[k] = new Mem[X]

Is write-back worth the trouble? Depends on (1) cost of write; (2) consistency issues.

---

**Write-back w/ “Dirty” bits**

<table>
<thead>
<tr>
<th>D</th>
<th>V</th>
<th>TAG</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>Mem[A]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>Mem[B]</td>
</tr>
</tbody>
</table>

ON REFERENCE TO Mem[X]: Look for X among tags...

**HIT:** X = TAG(i), for some cache line i

- **READ:** return DATA[i]
- **WRITE:** change DATA[i]; Start Write to Mem[X] D[i]=1

**MISS:** X not found in TAG of any cache line

- **REPLACEMENT SELECTION:**
  - Select some line k to hold Mem[X]
  - If D[k] == 1 (Write Back) Write Data(k) to Mem[Tag[k]]
- **READ:** Read Mem[X]
  - Set TAG[k] = X, DATA[k] = Mem[X], D[k]=0
- **WRITE:** Start Write to Mem[X]
  - Set TAG[k] = X, DATA[k] = new Mem[X]

---

**Caches: Summary**

**Associativity:**
- Less important as size increases
- 2-way or 4-way usually plenty for typical program clustering; BUT additional associativity
  - Smooths performance curve
  - Reduces number of select bits (we’ll see shortly how this helps)
- TREND: Invest in RAM, not comparators.

**Replacement Strategy:**
- BIG caches: any sane approach works well
- REAL randomness assuages paranoia!

**Performance analysis:**
- Tedious hand synthesis may build intuition from simple examples, BUT
- Computer simulation of cache behavior on REAL programs (or using REAL trace data) is the basis for most real-world cache design decisions.