CS 2630
Computer Organization

Meeting 28: Caches
Brandon Myers
University of Iowa
Memory hierarchy of a processor

- CPU (registers and logic): ~256 B
- Cache (SRAM): ~10 MB
- Main memory (DRAM): ~10 GB

Delay times:
- ~1 ns
- ~100 ns
Cache Performance
local shelves

→

central library (far away)

desk
Peer instruction

If you want to answer the question “How long will it take me to access my sources 1000 times?”

What parameters (variables) will you need to know?
Performance metrics

- *cache hit* latency: time for a cache hit
- *cache miss* latency: time for a cache miss
- *cache hit* ratio = cache hits / total accesses
  - or, cache miss ratio = 1 – cache hit ratio

Vocabulary
- cache hit: read or write data that is already in the cache
- cache miss: the data you want to read or write is not in the cache; must go to memory
What performance metrics can we influence?

• Library manager/designer controls
  • **hit latency** (how big is the local shelves? 1-10min?)
  • **miss latency** (how long do orders take? 1-7 days?)
    • within realistic ranges, we have some design control

Peer instruction: What determines the **hit ratio**?
parameters that affect hit ratio

• what is the capacity of the shelves?
• how many different books are we reading?
• what algorithm do we use to replace books on the shelf when it gets full?
• how many times to we access a book before it leaves the shelf to go back to central library?
• how well can we predict what books to order ahead of time? (librarian or student could be in charge of this task)
Memory hierarchy of a processor

Memory accesses first go to the cache. If the word is not in the cache then the processor **stalls** to wait for the word to be brought into cache

a subset of the words in memory can fit in the cache at any time
average memory access time (AMAT)

AMAT: expected time for a memory access

AMAT = hit latency + Miss rate x Miss latency
Peer instruction
Comparing performance 2 different caches

If we increase the capacity of our cache, what is likely to happen to each of the three terms?

\[ AMAT = \text{hit latency} + \text{miss rate} \times \text{miss latency} \]

(1) (2) (3)

a) increase
b) no change
c) decrease
Peer instruction
Comparing performance of a *processor* with those 2 different sized caches.

If we increase the capacity of our cache, what is likely to happen to each of the three factors?

\[
\frac{\text{seconds}}{\text{program}} = \frac{\text{seconds}}{\text{cycle}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{instructions}}{\text{program}}
\]

(1) (2) (3)

a) increase
b) no change
c) decrease
Cache Implementation
IBM Power 7
Memory hierarchy of a processor

- CPU (has registers)
- Cache
- Main memory

Memory accesses first go to the cache.

How do we check whether the word is in the cache?

A subset of the words in memory can fit in the cache at any time.
How do we check whether the word is in the cache?

Could store the address and the data in the cache (we’ll call the address a Tag since we physically store it as a label for the data)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x12340000</td>
<td>(4-byte word)</td>
</tr>
<tr>
<td>1</td>
<td>0x12340001</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>0x2001000e</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>0x20040000</td>
<td>...</td>
</tr>
</tbody>
</table>
How do we check whether the word is in the cache?

- Could *store* the address and the data in the cache (we’ll call the address a *Tag* since we physically store it as a label for the data)

read word at address 0x20040004

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x12340000</td>
</tr>
<tr>
<td>1</td>
<td>0x12340001</td>
</tr>
<tr>
<td>2</td>
<td>0x2001000e</td>
</tr>
<tr>
<td>3</td>
<td>0x20040000</td>
</tr>
</tbody>
</table>

MISS
to check if the data is in the cache:
- do an *associative search* of the Tags stored in the cache. If there is a match, then hit!, else miss!
How do we check whether the word is in the cache?

• Could *store* the address and the data in the cache (we’ll call the address a *Tag* since we physically store it as a label for the data)

read word at address 0x20040004

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x12340000</td>
</tr>
<tr>
<td>1</td>
<td>0x12340001</td>
</tr>
<tr>
<td>2</td>
<td>0x2001000e</td>
</tr>
<tr>
<td>3</td>
<td>0x20040000</td>
</tr>
</tbody>
</table>

• This cache implementation is expensive!
  • Time or chip area: Associative search requires *either* lots of circuitry (if checking all rows in parallel) or many cycles (if checking sequentially)
  • chip area: store a address bits (Tag) for every datum we store

Let’s tweak some design parameters to reduce the time or area
Design parameter #1: line size
Line size

- store the tag: 30 bits of address for every datum in the cache

<table>
<thead>
<tr>
<th></th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(30 bits)</td>
<td>(4 bytes)</td>
</tr>
<tr>
<td>2</td>
<td>(30 bits)</td>
<td>(4 bytes)</td>
</tr>
</tbody>
</table>

- If we must store 30 bits for each 1-word (4 bytes) datum, then there is a storage overhead of $30/32 = 94\%$.

- We can store more data per row. The amount of data is called the *line size*. If we make the line size 64 bytes, then the overhead is less because we only store one Tag per 64 bytes. E.g.,

<table>
<thead>
<tr>
<th></th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(?? bits)</td>
<td>(64 bytes)</td>
</tr>
<tr>
<td>1</td>
<td>(?? bits)</td>
<td>(64 bytes)</td>
</tr>
</tbody>
</table>
Peer instruction

How many bits of address do we need to store as the Tag for each line in this cache (assuming 32-bit addresses)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(?? bits)</td>
</tr>
<tr>
<td>1</td>
<td>(64 bytes)</td>
</tr>
</tbody>
</table>

(numeric response)
Peer instruction

If our cache contains $C$ bytes of data with $B$ bytes per line what is the total number of bits required to store all the tags? (assume byte-addressed memory and 32-bit addresses)
Lookups for larger line sizes

How do we perform lookups when the line size is larger than the size of the data being looked up?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (16 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0200100</td>
</tr>
<tr>
<td>1</td>
<td>0x1100204</td>
</tr>
<tr>
<td>2</td>
<td>0x2000001</td>
</tr>
<tr>
<td>3</td>
<td>0x1100044</td>
</tr>
</tbody>
</table>
Analogy: addresses on apartment buildings
Lookups for larger line sizes

• How do we perform lookups when the line size is larger than the size of the data being looked up?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (16 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0200100</td>
</tr>
<tr>
<td>1</td>
<td>0x1100204</td>
</tr>
<tr>
<td>2</td>
<td>0x2000001</td>
</tr>
<tr>
<td>3</td>
<td>0x1100044</td>
</tr>
</tbody>
</table>

1. Do the associative lookup: compare the upper 28 bits of the address with the Tags
2. If find a match, then hit! Now how do we find the byte we are looking for?
   • use the lower bits of the address as an **offset** into the line.
Example of a lookup

- architecture
  - byte-addressed
  - 12-bit addresses
- microarchitecture
  - cache line size = 8 bytes

Implications
- 3 bits of offset, to choose between the 8 bytes per line
- 9 bits of tag (the remaining bits)

Read byte at address 0x123

```
0001 0010 0011
```

<table>
<thead>
<tr>
<th>tag</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0010 0100 (0x024)</td>
<td>011 (0x3)</td>
</tr>
</tbody>
</table>

Tag Data (8 bytes)

<table>
<thead>
<tr>
<th></th>
<th>Data (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Hit!

Step 1: search for this tag among the tags stored in the cache

Step 2: read the byte at the offset
Peer instruction

- architecture
  - byte-addressed
  - 10-bit addresses
- microarchitecture
  - cache line size = 4 bytes

1. fill in tag and offset
2. which byte in the cache is accessed?

read byte at address 0x2D2

<table>
<thead>
<tr>
<th>tag</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (4 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4C</td>
<td>1 2 3</td>
</tr>
<tr>
<td>0x6A</td>
<td></td>
</tr>
<tr>
<td>0xD3</td>
<td></td>
</tr>
<tr>
<td>0xB4</td>
<td></td>
</tr>
</tbody>
</table>
Peer instruction

Comparing performance of 2 different caches

If we keep the capacity of our cache constant while increasing line size, how will the terms change?

\[ AMAT = \text{hit latency} + \text{miss rate} \times \text{miss latency} \]

\begin{align*}
(1) & \quad (2) & \quad (3)
\end{align*}

a) increase  
b) no change  
c) decrease
• This cache implementation is expensive!
  
  - Time or chip area: Associative search requires *either* lots of circuitry (if checking all rows in parallel) or many cycles (if checking sequentially)

✓ chip area: store 4 bytes of address for every datum we store

increase *line size* to reduce how much of the address we need to store
Design parameter #2: *associativity*
associativity

• the caches we’ve looked at so far are “fully associative”
  • analogy: a fully associative cache is like a hash table with a single bucket; we need to check all the elements of the bucket for our tag (key)
associativity

• the caches we’ve looked at so far are “fully associative”
  • analogy: a fully associative cache is like a hash table with a single bucket; we need to check all the elements of the bucket for our tag (key)

• We can **reduce the associativity** to restrict the number of places that a given cache line is allowed to be put

![Diagram of full associative cache and 2-way set associative cache]
Example of a lookup: 2 way set-associative cache

- architecture
  - byte-addressed
  - 13-bit addresses
- microarchitecture
  - cache line size = 8 bytes

**implications**
- 3 bits of offset, to choose between the 8 bytes per line
- 2 bits of index to choose the set
- 8 bits of tag (the remaining bits)

read word at address 0x012E

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001 0010 (0x12)</td>
<td>1</td>
<td>110   (0x6)</td>
</tr>
</tbody>
</table>

step 1: search for this tag among the tags stored in set 1

hit!

step 2: read the byte at the offset

- architecture
- byte-addressed
- 13-bit addresses
- microarchitecture
  - cache line size = 8 bytes

- 3 bits of offset, to choose between the 8 bytes per line
- 2 bits of index to choose the set
- 8 bits of tag (the remaining bits)
- Time or chip area: Associative search requires *either* lots of circuitry (if checking all rows in parallel) or many cycles (if checking sequentially)

A smaller associativity means we check fewer tags per lookup

For example, this 2-way set associative cache has 6 total lines, but you only have to check the tags in 2 of them (the 2 ways of set 1)

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0  1  2  3  4  5  6  7</td>
</tr>
<tr>
<td>0</td>
<td>0x12</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x60</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0x99</td>
<td></td>
</tr>
</tbody>
</table>

Note: The remaining lines in the cache are not shown.
Peer instruction
- architecture
  - byte-addressed
  - 12-bit addresses
- microarchitecture
  - cache line size = 8 bytes

read word at address 0x22B

1. fill in tag, index, offset
2. which byte in the cache is accessed?

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Data (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0x11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x34</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x22</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x60</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x22</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x1C</td>
<td></td>
</tr>
</tbody>
</table>
Peer instruction

For fixed capacity and line size, how does increasing associativity change AMAT?

AMAT = hit latency + miss rate x miss latency

give three answers
hit latency, miss rate, miss latency

A: Increases hit time, decreases miss rate
B: Decreases hit time, decreases miss rate
C: Increases hit time, increases miss rate
D: Decreases hit time, increases miss rate

(direct-mapped or 1-line per set is lowest associativity
fully associative or all lines in 1 set is highest associativity)
Peer instruction

For fixed total cache capacity and associativity, how does \textit{larger line size} change AMAT?
AMAT = hit latency + miss rate \times miss latency

give three answers
hit latency, miss rate, miss latency

A: decrease
B: no change
C: increase
D: can’t determine
What happens when the data isn’t in the cache? (cache miss)
What happens when we miss?

reading address 0x414.  (0100 0001 0000)

<table>
<thead>
<tr>
<th>tag</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1000 0010 (0x082)</td>
<td>100 (0x4)</td>
</tr>
</tbody>
</table>

no match in the tags

**MISS!**

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x111</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>0x020</td>
<td></td>
</tr>
<tr>
<td>0x024</td>
<td></td>
</tr>
<tr>
<td>0x003</td>
<td></td>
</tr>
</tbody>
</table>

main memory
What happens when we miss?

reading address 0x414.  (0100 0001 0100)

<table>
<thead>
<tr>
<th>tag</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1000 0010 (0x082)</td>
<td>100 (0x4)</td>
</tr>
</tbody>
</table>

no match in the tags

MISS!

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data (8 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x111</td>
</tr>
<tr>
<td>1</td>
<td>0x020</td>
</tr>
<tr>
<td>2</td>
<td>0x024</td>
</tr>
<tr>
<td>3</td>
<td>0x003</td>
</tr>
</tbody>
</table>

main memory

find the 8 bytes starting at 0x410  
(why 0x410 and not 0x414?)

read the data into the cache, replacing some other line; update the tag
Cache line replacement

• How do we pick the line (victim) that will be kicked out?

• Cache replacement policy

• Examples
  • least recently used (LRU) – victim is the line last accessed by the CPU the longest time ago
  • approximate LRU – approximation of LRU
  • least frequently used (LFU) – victim is the line that has been accessed the least
  • Random – pick the victim (pseudo)-randomly
  • many others!

• How do we measure how good a replacement policy is?
• Why would LRU be a good policy?
Misses (3 C’s)

• Compulsory: miss because it is the first time you bring it into the cache

• Capacity: miss specifically because the cache was full

• Conflict: miss because specifically because it got kicked out by another line
Misses (3 C’s)

• Compulsory: miss because it is the first time you bring it into the cache

  cache: fully associative, 2 lines, LRU replacement

  compulsory

Accesses over time
**Misses (3 C’s)**

- **Capacity**: miss specifically because the cache was full

  cache: fully associative, 2 lines, LRU replacement

  compulsory

  capacity

  *least recently used (LRU)* – victim is the line last accessed by the CPU the longest time ago
Misses (3 C’s)

- Conflict: miss because specifically because it got kicked out by another line
  cache: 2-way set associative, 2 sets, LRU replacement
In the same capacity 4-line fully associative cache, the red block would have been kicked out for the purple block.
Peer instruction

For fixed capacity and line size, how does increasing **associativity** change the % of misses each type of miss contributes?

Compulsory, Capacity, Conflict

A: decrease
B: no change
C: increase
D: can’t determine

(direct-mapped or 1-line per set is lowest associativity
fully associative or all lines in 1 set is highest associativity)