CSE 410
Computer Systems

Hal Perkins
Spring 2010
Lecture 12 – More About Caches
Reading

• Computer Organization and Design
  – Section 5.1 Introduction
  – Section 5.2 Basics of Caches
  – Section 5.3 Measuring and Improving Cache Performance
How big is the cache?

Suppose we have a byte-addressable machine with 16-bit addresses with a cache with the following characteristics:

• It is direct-mapped
• Each block holds one byte
• The cache index is the four least significant bits

Two questions:
• How many blocks does the cache hold?
• How many bits of storage are required to build the cache (e.g., for the data array, tags, etc.)?
More cache organizations

Now we’ll explore some alternate cache organizations.
  – How can we take advantage of spatial locality too?
  – How can we reduce the number of potential conflicts?

• First, a brief discussion about cache performance.
Memory System Performance

• To examine the performance of a memory system, we need to focus on a couple of important factors.
  – How long does it take to send data from the cache to the CPU?
  – How long does it take to copy data from memory into the cache?
  – How often do we have to access main memory?

• There are names for all of these variables.
  – The **hit time** is how long it takes data to be sent from the cache to the processor. This is usually fast, on the order of 1-3 clock cycles.
  – The **miss penalty** is the time to copy data from main memory to the cache. This often requires dozens of clock cycles (at least).
  – The **miss rate** is the percentage of misses.
Average memory access time

• The average memory access time, or AMAT, can then be computed.

\[
\text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty})
\]

This is just averaging the amount of time for cache hits and the amount of time for cache misses.

• How can we improve the average memory access time of a system?
  – Obviously, a lower AMAT is better.
  – Miss penalties are usually much greater than hit times, so the best way to lower AMAT is to reduce the miss penalty or the miss rate.

• However, AMAT should only be used as a general guideline. Remember that execution time is still the best performance metric.
Performance example

• Assume data accesses only. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

\[
\text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty}) \\
= 1 \text{ cycle} + (3\% \times 20 \text{ cycles}) \\
= 1.6 \text{ cycles}
\]

• If the cache was perfect and never missed, the AMAT would be one cycle. But even with just a 3% miss rate, the AMAT here increases 1.6 times!

• How can we reduce miss rate?
Spatial locality

- One-byte cache blocks don’t take advantage of spatial locality, which predicts that an access to one address will be followed by an access to a nearby address.
- What can we do?
Spatial locality

- What we can do is make the cache block size larger than one byte.
- Here we use two-byte blocks, so we can load the cache with two bytes at a time.
- If we read from address 12, the data in addresses 12 and 13 would both be copied to cache block 2.
Block addresses

- Now how can we figure out where data should be placed in the cache?
- It’s time for block addresses! If the cache block size is $2^n$ bytes, we can conceptually split the main memory into $2^n$-byte chunks too.
- To determine the block address of a byte address $i$, you can do the integer division

$$i / 2^n$$

- Our example has two-byte cache blocks, so we can think of a 16-byte main memory as an “8-block” main memory instead.
- For instance, memory addresses 12 and 13 both correspond to block address 6, since $12 / 2 = 6$ and $13 / 2 = 6$. 

![Diagram showing block addresses]
Cache mapping

Once you know the block address, you can map it to the cache as before: find the remainder when the block address is divided by the number of cache blocks.

In our example, memory block 6 belongs in cache block 2, since $6 \mod 4 = 2$.

This corresponds to placing data from memory byte addresses 12 and 13 into cache block 2.
Data placement within a block

- When we access one byte of data in memory, we’ll copy its entire block into the cache, to hopefully take advantage of spatial locality.
- In our example, if a program reads from byte address 12 we’ll load all of memory block 6 (both addresses 12 and 13) into cache block 2.
- Note byte address 13 corresponds to the same memory block address! So a read from address 13 will also cause memory block 6 (addresses 12 and 13) to be loaded into cache block 2.
- To make things simpler, byte i of a memory block is always stored in byte i of the corresponding cache block.
Locating data in the cache

- Let's say we have a cache with $2^k$ blocks, each containing $2^n$ bytes.
- We can determine where a byte of data belongs in this cache by looking at its address in main memory.
  - $k$ bits of the address will select one of the $2^k$ cache blocks.
  - The lowest $n$ bits are now a block offset that decides which of the $2^n$ bytes in the cache block will store the data.

Our example used a $2^2$-block cache with $2^1$ bytes per block. Thus, memory address 13 (1101) would be stored in byte 1 of cache block 2.
A picture (4-bit addresses!)
An exercise

For the addresses below, what byte is read from the cache (or is there a miss)?

- 1010
- 1110
- 0001
- 1101
Using arithmetic

• An equivalent way to find the right location within the cache is to use arithmetic again.

• We can find the index in two steps, as outlined earlier.
  – Do integer division of the address by $2^n$ to find the block address.
  – Then mod the block address with $2^k$ to find the index.
  
• The block offset is just the memory address mod $2^n$.
• For example, we can find address 13 in a 4-block, 2-byte per block cache.
  – The block address is $13 / 2 = 6$, so the index is then $6 \mod 4 = 2$.
  – The block offset would be $13 \mod 2 = 1$. 
Here is a cache with 1,024 blocks of 4 bytes each, and 32-bit memory addresses.
A larger example cache mapping

• Where would the byte from memory address 6146 be stored in this direct-mapped $2^{10}$-block cache with $2^{2}$-byte blocks?

• We can determine this with the binary force.
  – 6146 in binary is 00...01 1000 0000 00 10.
  – The lowest 2 bits, 10, mean this is the second byte in its block.
  – The next 10 bits, 1000000000, are the block number itself (512).

• Equivalently, you could use arithmetic instead.
  – The block offset is 6146 mod 4, which equals 2.
  – The block address is $6146/4 = 1536$, so the index is $1536 \mod 1024$, or 512.
A larger diagram of a larger example cache mapping

Address (32 bits)

0000 .... 0001 1000000000 10

2 bits

Index  Valid  Tag  Data

0      
1      
2      
...    
512    
...    
1022   
1023   

Tag  =  Mux

Hit  Data
What goes in the rest of that cache block?

- The other three bytes of that cache block come from the same memory block, whose addresses must all have the same index (1000000000) and the same tag (00...01).

```
Address (32 bits)
0000 .... 0001 1000000000 10
```

```
Index  Valid  Tag  Data
...    ...  ... ...

Mux  Hit  Data
```
The rest of that cache block

- Again, byte $i$ of a memory block is stored into byte $i$ of the corresponding cache block.
  - In our example, memory block 1536 consists of byte addresses 6144 to 6147. So bytes 0-3 of the cache block would contain data from address 6144, 6145, 6146 and 6147 respectively.
  - You can also look at the lowest 2 bits of the memory address to find the block offsets.

<table>
<thead>
<tr>
<th>Block offset</th>
<th>Memory address</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00..01 1000000000 00</td>
<td>6144</td>
</tr>
<tr>
<td>01</td>
<td>00..01 1000000000 01</td>
<td>6145</td>
</tr>
<tr>
<td>10</td>
<td>00..01 1000000000 10</td>
<td>6146</td>
</tr>
<tr>
<td>11</td>
<td>00..01 1000000000 11</td>
<td>6147</td>
</tr>
</tbody>
</table>

Index | Valid | Tag | Data |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>512</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Disadvantage of direct mapping

- The direct-mapped cache is easy: indices and offsets can be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block.
- However, this isn’t really flexible. If a program uses addresses 2, 6, 2, 6, 2, ..., then each access will result in a cache miss and a load into cache block 2.
- This cache has four blocks, but direct mapping might not let us use all of them.
- This can result in more misses than we might like.
A fully associative cache

- A fully associative cache permits data to be stored in any cache block, instead of forcing each memory address into one particular block.
  - When data is fetched from memory, it can be placed in any unused block of the cache.
  - This way we’ll never have a conflict between two or more memory addresses which map to a single cache block.
- In the previous example, we might put memory address 2 in cache block 2, and address 6 in block 3. Then subsequent repeated accesses to 2 and 6 would all be hits instead of misses.
- If all the blocks are already in use, it’s usually best to replace the least recently used one, assuming that if it hasn’t used it in a while, it won’t be needed again anytime soon.
The price of full associativity

- However, a fully associative cache is expensive to implement.
  - Because there is no index field in the address anymore, the *entire* address must be used as the tag, increasing the total cache size.
  - Data could be anywhere in the cache, so we must check the tag of *every* cache block. That’s a lot of hardware! (Expensive and starting to get slow)
Set associativity

- An intermediate possibility is a set-associative cache.
  - The cache is divided into groups of blocks, called sets.
  - Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.
  - Each block in the set has a separate tag
- If each set has \(2^x\) blocks, the cache is an \(2^x\)-way associative cache.
- Here are several possible organizations of an eight-block cache.
Locating a set associative block

• We can determine where a memory address belongs in an associative cache in a similar way as before.

• If a cache has $2^s$ sets and each block has $2^n$ bytes, the memory address can be partitioned as follows.

  Address (m bits)  
  \[ \text{Tag} \quad \text{Index} \quad \text{Block offset} \]  

  \[ \text{(m-s-n)} \quad s \quad n \]  

• Our arithmetic computations now compute a set index, to select a set within the cache instead of an individual block.

  \[ \begin{align*}  
  \text{Block Offset} & = \text{Memory Address mod } 2^n \\  
  \text{Block Address} & = \text{Memory Address} / 2^n \\  
  \text{Set Index} & = \text{Block Address mod } 2^s 
  \end{align*} \]

• The tag is compared to all of the tags in that cache set to see which block (if any) in that set is a match.
Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in any block, shown in green, within the correct set.
Block replacement

- Any empty block in the correct set may be used for storing data.
- If there are no empty blocks, the cache controller will attempt to replace the least recently used block, just like before.
- For highly associative caches, it’s expensive to keep track of what’s really the least recently used block, so some approximations are used. We won’t get into the details.

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Sets</th>
<th>Blocks per Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-way</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>2-way</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>4-way</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary

- Larger block sizes can take advantage of spatial locality by loading data from not just one address, but also nearby addresses, into the cache.
- **Associative caches** assign each memory address to a particular set within the cache, but not to any specific block within that set.
  - Set sizes range from 1 (direct-mapped) to $2^k$ (fully associative).
  - Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost.
  - In practice, 2-way through 16-way set-associative caches strike a good balance between lower miss rates and higher costs.
- Next, we’ll talk more about measuring cache performance, and also discuss the issue of writing data to a cache.