Today:
- Quick Review
- LRU
- Writes
- Cache performance?
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.
- But, what happens if a program uses addresses 2, 6, 2, 6, 2, ...?

How do we solve this problem?
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.
- 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.

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

![Diagram of set associativity](image)
Quick exercise *(hint hint)*

- Block size = 16 bytes
- Where would these addresses go?
  - 0, 16, 32, 64, 128
- How can you figure out block size?
- What about associativity?
- What about cache size?

2-way associativity
4 sets, 2 blocks each

Set

0

1

2

3
### 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>1-way associativity</th>
<th>2-way associativity</th>
<th>4-way associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 sets, 1 block each</td>
<td>4 sets, 2 blocks each</td>
<td>2 sets, 4 blocks each</td>
</tr>
</tbody>
</table>

```
Set
0
1
2
3
4
5
6
7
```
```
Set
0
1
2
3
```
```
Set
0
1
```

LRU example

- Assume a fully-associative cache with two blocks, which of the following memory references miss in the cache.
  - assume distinct addresses go to distinct blocks

<table>
<thead>
<tr>
<th>addresses</th>
<th>0</th>
<th>Tags</th>
<th>1</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td>--</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>--</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2-way set associative cache implementation

- How does an implementation of a 2-way cache compare with that of a fully-associative cache?

- Only two comparators are needed.
- The cache tags are a little shorter too.
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.
Cache Writing & Performance

- We’ll now cover:
  - Writing to caches: keeping memory consistent & write-allocation.
  - We’ll try to quantify the benefits of different cache designs, and see how caches affect overall performance.
  - We’ll also investigate some main memory organizations that can help increase memory system performance.
- Next, we’ll talk about Virtual Memory, where memory is treated like a cache of the disk.
Four important questions

1. When we copy a block of data from main memory to the cache, where exactly should we put it?

2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we’d have to replace one of the existing blocks in the cache... which one?

4. How can write operations be handled by the memory system?

- We’ve answered the first 3. Now, we consider the 4th.
Writing to a cache

- Writing to a cache raises several additional issues.
- First, let’s assume that the address we want to write to is already loaded in the cache. We’ll assume a simple direct-mapped cache.

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11010</td>
<td>42803</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>110 0110</td>
<td>42803</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- If we write a new value to that address, we can store the new data in the cache, and avoid an expensive main memory access.

\[ \text{Mem}[214] = 21763 \]

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>110 0110</td>
<td>42803</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Inconsistent memory

- But now the cache and memory contain different, inconsistent data!
- How can we ensure that subsequent loads will return the right value?
- This is also problematic if other devices are sharing the main memory, as in a multiprocessor system.

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>110 0110</td>
<td>42803</td>
</tr>
</tbody>
</table>
A write-through cache solves the inconsistency problem by forcing all writes to update both the cache and the main memory.

This is simple to implement and keeps the cache and memory consistent.

Why is this not so good?
Write-back caches

- In a write-back cache, the memory is not updated until the cache block needs to be replaced (e.g., when loading data into a full cache set).
- For example, we might write some data to the cache at first, leaving it inconsistent with the main memory as shown before. The cache block is marked “dirty” to indicate this inconsistency.

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem[214] = 21763</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 1110</td>
<td>1225</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 0110</td>
<td>42803</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Subsequent reads to the same memory address will be serviced by the cache, which contains the correct, updated data.
Finishing the write back

- We don’t need to store the new value back to main memory unless the cache block gets replaced.
- For example, on a read from Mem[142], which maps to the same cache block, the modified cache contents will first be written to main memory.

```
<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
<td>1101 0110</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- Only then can the cache block be replaced with data from address 142.

```
<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>0</td>
<td>10001</td>
<td>1225</td>
<td>1101 0110</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Write-back cache discussion

- Each block in a write-back cache needs a **dirty bit** to indicate whether or not it must be saved to main memory before being replaced—otherwise we might perform unnecessary writebacks.

- Notice the penalty for the main memory access will not be applied until the execution of some *subsequent* instruction following the write.
  - In our example, the write to Mem[214] affected only the cache.
  - But the load from Mem[142] resulted in *two* memory accesses: one to save data to address 214, and one to load data from address 142.
    - The write can be “buffered” as was shown in write-through.

- The advantage of write-back caches is that not all write operations need to access main memory, as with write-through caches.
  - If a single address is frequently written to, then it doesn’t pay to keep writing that data through to main memory.
  - If several bytes within the same cache block are modified, they will only force one memory write operation at write-back time.
Write misses

- A second scenario is if we try to write to an address that is not already contained in the cache; this is called a write miss.
- Let’s say we want to store 21763 into Mem[1101 0110] but we find that address is not currently in the cache.

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>00010</td>
<td>123456</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1101 0110</td>
<td>6378</td>
</tr>
</tbody>
</table>

- When we update Mem[1101 0110], should we also load it into the cache?
With a **write around** policy, the write operation goes directly to main memory *without* affecting the cache.

![Index Tag Data Address Data](image)

Mem[214] = 21763
Write around caches (a.k.a. write-no-allocate)

- With a write around policy, the write operation goes directly to main memory without affecting the cache.

- This is good when data is written but not immediately used again, in which case there’s no point to load it into the cache yet.

```
for (int i = 0; i < SIZE; i++)
    a[i] = i;
```
### Allocate on write

- An *allocate on write* strategy would instead load the newly written data into the cache.

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>11010</td>
<td>21763</td>
<td>1101010</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- If that data is needed again soon, it will be available in the cache.
Which is it?

- Given the following trace of accesses, can you determine whether the cache is **write-allocate** or **write-no-allocate**?
  - Assume A and B are distinct, and can be in the cache simultaneously.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Miss</strong></td>
<td>Load A</td>
</tr>
<tr>
<td><strong>Miss</strong></td>
<td>Store B</td>
</tr>
<tr>
<td><strong>Hit</strong></td>
<td>Store A</td>
</tr>
<tr>
<td><strong>Hit</strong></td>
<td>Load A</td>
</tr>
<tr>
<td><strong>Miss</strong></td>
<td>Load B</td>
</tr>
<tr>
<td><strong>Hit</strong></td>
<td>Load B</td>
</tr>
<tr>
<td><strong>Hit</strong></td>
<td>Load A</td>
</tr>
</tbody>
</table>
Which is it?

- Given the following trace of accesses, can you determine whether the cache is **write-allocate** or **write-no-allocate**?
  - Assume A and B are distinct, and can be in the cache simultaneously.

<table>
<thead>
<tr>
<th>Event</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Miss</td>
<td>Load A</td>
</tr>
<tr>
<td>Miss</td>
<td>Store B</td>
</tr>
<tr>
<td>Hit</td>
<td>Store A</td>
</tr>
<tr>
<td>Hit</td>
<td>Load A</td>
</tr>
<tr>
<td>Miss</td>
<td>Load B</td>
</tr>
<tr>
<td>Hit</td>
<td>Load B</td>
</tr>
<tr>
<td>Hit</td>
<td>Load A</td>
</tr>
</tbody>
</table>

Answer: Write-no-allocate

On a write-allocate cache this would be a hit
First Observations

- **Split Instruction/Data caches:**
  - Pro: No structural hazard between IF & MEM stages
    - A single-ported unified cache stalls fetch during load or store
  - Con: Static partitioning of cache between instructions & data
    - Bad if working sets unequal: e.g., code/\texttt{DATA} or \texttt{CODE}/data

- **Cache Hierarchies:**
  - Trade-off between access time & hit rate
    - L1 cache can focus on fast access time (okay hit rate)
    - L2 cache can focus on good hit rate (okay access time)
  - Such hierarchical design is another “big idea”
  - We’ll see this in section.
Opteron Vital Statistics

- **L1 Caches: Instruction & Data**
  - 64 kB
  - 64 byte blocks
  - 2-way set associative
  - 2 cycle access time

- **L2 Cache:**
  - 1 MB
  - 64 byte blocks
  - 4-way set associative
  - 16 cycle access time (total, not just miss penalty)

- **Memory**
  - 200+ cycle access time
Comparing cache organizations

- Like many architectural features, caches are evaluated experimentally.
  - As always, performance depends on the actual instruction mix, since different programs will have different memory access patterns.
  - Simulating or executing real applications is the most accurate way to measure performance characteristics.
- The graphs on the next few slides illustrate the simulated miss rates for several different cache designs.
  - Again lower miss rates are generally better, but remember that the miss rate is just one component of average memory access time and execution time.
  - You’ll probably do some cache simulations if you take CS433.
Associativity tradeoffs and miss rates

- As we saw last time, higher associativity means more complex hardware.
- But a highly-associative cache will also exhibit a lower miss rate.
  - Each set has more blocks, so there’s less chance of a conflict between two addresses which both belong in the same set.
  - Overall, this will reduce AMAT and memory stall cycles.
- The textbook shows the miss rates decreasing as the associativity increases.
Cache size and miss rates

- The cache size also has a significant impact on performance.
  - The larger a cache is, the less chance there will be of a conflict.
  - Again this means the miss rate decreases, so the AMAT and number of memory stall cycles also decrease.
- The complete Figure 7.29 depicts the miss rate as a function of both the cache size and its associativity.
Block size and miss rates

- Finally, Figure 7.12 on p. 559 shows miss rates relative to the block size and overall cache size.
  - Smaller blocks do not take maximum advantage of spatial locality.
Memory and overall performance

- How do cache hits and misses affect overall system performance?
  - Assuming a hit time of one CPU clock cycle, program execution will continue normally on a cache hit. (Our earlier computations always assumed one clock cycle for an instruction fetch or data access.)
  - For cache misses, we’ll assume the CPU must stall to wait for a load from main memory.

- The total number of stall cycles depends on the number of cache misses and the miss penalty.

  Memory stall cycles = Memory accesses x miss rate x miss penalty

- To include stalls due to cache misses in CPU performance equations, we have to add them to the “base” number of execution cycles.

  CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time
Performance example

- Assume that 33% of the instructions in a program are data accesses. The cache hit ratio is 97% and the hit time is one cycle, but the miss penalty is 20 cycles.

\[
\text{Memory stall cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty} \\
= 0.33 \times I \times 0.03 \times 20 \text{ cycles} \\
= 0.2 I \text{ cycles}
\]

- If \( I \) instructions are executed, then the number of wasted cycles will be \( 0.2 \times I \).

This code is 1.2 times slower than a program with a “perfect” CPI of 1!
Memory systems are a bottleneck

CPU time = (CPU execution cycles + Memory stall cycles) x Cycle time

- Processor performance traditionally outpaces memory performance, so the memory system is often the system bottleneck.
- For example, with a base CPI of 1, the CPU time from the last page is:

  \[ \text{CPU time} = (I + 0.2I) \times \text{Cycle time} \]

- What if we could \textit{double} the CPU performance so the CPI becomes 0.5, but memory performance remained the same?

  \[ \text{CPU time} = (0.5I + 0.2I) \times \text{Cycle time} \]

- The overall CPU time improves by just \(1.2/0.7 = 1.7\) times!
- Refer back to Amdahl’s Law from textbook page 101.
  - Speeding up only part of a system has diminishing returns.
Basic main memory design

- There are some ways the main memory can be organized to reduce miss penalties and help with caching.
- For some concrete examples, let’s assume the following three steps are taken when a cache needs to load data from the main memory.
  1. It takes 1 cycle to send an address to the RAM.
  2. There is a 15-cycle latency for each RAM access.
  3. It takes 1 cycle to return data from the RAM.
- In the setup shown here, the buses from the CPU to the cache and from the cache to RAM are all one word wide.
- If the cache has one-word blocks, then filling a block from RAM (i.e., the miss penalty) would take 17 cycles.
  
  \[1 + 15 + 1 = 17 \text{ clock cycles}\]
- The cache controller has to send the desired address to the RAM, wait and receive the data.
Miss penalties for larger cache blocks

- If the cache has four-word blocks, then loading a single block would need four individual main memory accesses, and a miss penalty of 68 cycles!

\[4 \times (1 + 15 + 1) = 68 \text{ clock cycles}\]
A wider memory

- A simple way to decrease the miss penalty is to widen the memory and its interface to the cache, so we can read multiple words from RAM in one shot.
- If we could read four words from the memory at once, a four-word cache load would need just 17 cycles.

\[1 + 15 + 1 = 17\] cycles

- The disadvantage is the cost of the wider buses—each additional bit of memory width requires another connection to the cache.
An interleaved memory

- Another approach is to **interleave** the memory, or split it into “banks” that can be accessed individually.
- The main benefit is overlapping the latencies of accessing each word.
- For example, if our main memory has four banks, each one byte wide, then we could load four bytes into a cache block in just 20 cycles.

  \[1 + 15 + (4 \times 1) = 20\]  

- Our buses are still one byte wide here, so four cycles are needed to transfer data to the caches.
- This is cheaper than implementing a four-byte bus, but not too much slower.
Here is a diagram to show how the memory accesses can be interleaved.

- The magenta cycles represent sending an address to a memory bank.
- Each memory bank has a 15-cycle latency, and it takes another cycle (shown in blue) to return data from the memory.

This is the same basic idea as pipelining!

- As soon as we request data from one memory bank, we can go ahead and request data from another bank as well.
- Each individual load takes 17 clock cycles, but four overlapped loads require just 20 cycles.
Which is better?

- Increasing block size can improve hit rate (due to spatial locality), but transfer time increases. Which cache configuration would be better?

<table>
<thead>
<tr>
<th></th>
<th>Cache #1</th>
<th>Cache #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>32-bytes</td>
<td>64-bytes</td>
</tr>
<tr>
<td>Miss rate</td>
<td>5%</td>
<td>4%</td>
</tr>
</tbody>
</table>

- Assume both caches have single cycle hit times. Memory accesses take 15 cycles, and the memory bus is 8-bits wide:
  - i.e., an 16-byte memory access takes 18 cycles:
    - 1 (send address) + 15 (memory access) + 2 (two 8-byte transfers)

recall: \[ \text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty}) \]
Which is better?

- Increasing block size can improve hit rate (due to spatial locality), but transfer time increases. Which cache configuration would be better?

<table>
<thead>
<tr>
<th></th>
<th>Cache #1</th>
<th>Cache #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td>32-bytes</td>
<td>64-bytes</td>
</tr>
<tr>
<td>Miss rate</td>
<td>5%</td>
<td>4%</td>
</tr>
</tbody>
</table>

- Assume both caches have single cycle hit times. Memory accesses take 15 cycles, and the memory bus is 8-bytes wide:
  - i.e., an 16-byte memory access takes 18 cycles:
    - 1 (send address) + 15 (memory access) + 2 (two 8-byte transfers)

  **Cache #1:**
  
  Miss Penalty = 1 + 15 + 32B/8B = 20 cycles
  
  AMAT = 1 + (.05 * 20) = 2

  **Cache #2:**

  Miss Penalty = 1 + 15 + 64B/8B = 24 cycles
  
  AMAT = 1 + (.04 * 24) = ~1.96

  recall: \( \text{AMAT} = \text{Hit time} + (\text{Miss rate} \times \text{Miss penalty}) \)
Summary

- Writing to a cache poses a couple of interesting issues.
  - **Write-through** and **write-back** policies keep the cache consistent with main memory in different ways for write hits.
  - **Write-around** and **allocate-on-write** are two strategies to handle write misses, differing in whether updated data is loaded into the cache.

- Memory system performance depends upon the cache **hit time**, **miss rate** and **miss penalty**, as well as the actual program being executed.
  - We can use these numbers to find the **average memory access time**.
  - We can also revise our CPU time formula to include **stall cycles**.

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

\[
\text{Memory stall cycles} = \text{Memory accesses} \times \text{miss rate} \times \text{miss penalty}
\]

\[
\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}
\]

- The organization of a memory system affects its performance.
  - The cache size, block size, and associativity affect the miss rate.
  - We can organize the main memory to help reduce miss penalties. For example, **interleaved memory** supports pipelined data accesses.