CSE 410
Computer Systems

Hal Perkins
Spring 2010
Lecture 13 – Cache Writes and Performance
Reading

• Computer Organization and Design
  – Section 5.1 Introduction
  – Section 5.2 Basics of Caches
  – Section 5.3 Measuring and Improving Cache Performance
Cache Writing & Performance

• What’s left?
  – Writing to caches: keeping memory consistent & write-allocation.
  – We’ll also investigate some main memory organizations that can help increase memory system performance.
• Later, 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>
<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>42803</td>
<td>1101 0110</td>
<td>42803</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<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.

\[
\text{Mem}[214] = 21763
\]
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>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>1101 0110</td>
<td>42803</td>
</tr>
</tbody>
</table>
Write-through caches

- A write-through cache solves the inconsistency problem by forcing all writes to update both the cache and the main memory.

  ![Cache Diagram]

- 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

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

<table>
<thead>
<tr>
<th>Index</th>
<th>V</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tr>
<tr>
<td>...</td>
<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>1000 1110</td>
<td>1225</td>
</tr>
<tr>
<td>1101 0110</td>
<td>42803</td>
</tr>
<tr>
<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

• 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-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.
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>110</td>
<td>1</td>
<td>00010</td>
<td>123456</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?
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.

  ![Memory Table]

  ```
  for (int i = 0; i < SIZE; i++)
    a[i] = i;
  ```

- 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.

  ![Memory Diagram]
Allocate on write

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

\[
\begin{array}{|c|c|c|}
\hline
\text{Index} & \text{V} & \text{Tag} & \text{Data} \\
\hline
\ldots & \ldots & \ldots & \ldots \\
110 & 1 & 11010 & 21763 \\
\ldots & \ldots & \ldots & \ldots \\
\hline
\end{array}
\]

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

\[1 + 15 + (4 \times 1) = 20\text{ cycles}\]

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.
Interleaved memory accesses

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.
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.