Lectures 19 (maybe 20 too 😅)

- Today:
  - Writes
  - Cache performance?

Mystery caches!

Hmm, what about writes?

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 cache 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>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>42080</td>
<td>...</td>
<td>42080</td>
</tr>
<tr>
<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.

<table>
<thead>
<tr>
<th>Index</th>
<th>Tag</th>
<th>Data</th>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>0</td>
<td>256</td>
<td>1101010</td>
<td>256</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<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 they may each have different data.

Write-through caches

- 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.

- 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 address 1100, which maps to the same cache block, the modified cache contents will first be written to main memory.

- Only then can the cache block be replaced with data from address 142.
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 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 17753 into item [1001 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>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0000</td>
<td>123456</td>
<td>1101 0110</td>
<td>17753</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0000</td>
<td>123456</td>
<td>1101 0110</td>
<td>17753</td>
</tr>
</tbody>
</table>

- When we update item [1001 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.

allocate on write

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

allocate on write

- If that data is not yet loaded, 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.

<table>
<thead>
<tr>
<th>Hit</th>
<th>Store A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit</td>
<td>Store B</td>
</tr>
<tr>
<td>Miss</td>
<td>Load A</td>
</tr>
<tr>
<td>Miss</td>
<td>Load B</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 & ID/EX stages
  - Con: A single-port unified cache stalls fetch during load or store
    - Bad if working sets unequal: e.g., caches/data or code/...

- 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”

Opteron Vital Statistics

- L1 Caches: Instruction & Data
  - 4-way set associative
  - 1 cycle access time

- L2 Cache
  - 1 MB
  - 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 CSE 471.

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, 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 graph depicts the miss rate as a function of both the cache size and its associativity.
Block size and miss rates

- Finally, this graph shows miss rates relative to the block size and overall cache size.
  - Smaller blocks do not take maximum advantage of spatial locality.
  - Larger blocks do not take maximum advantage of spatial locality.

Memory and overall performance

- How do cache hit and miss rates 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 hits and the miss penalty.

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

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

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

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 10 cycles.
  \[
  \text{Memory stall cycles} = \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty} = 0.33 \times 0.03 \times 10 \times \text{cycles}
  \]
  \[
  = 0.1 \text{ cycles}
  \]

- If 1 instruction is executed, then the number of wasted cycles will be

  \[
  = 0.2 \times 1
  \]

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

Memory systems are a bottleneck

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

- 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:

  \[
  \text{CPU time} = (1 + 0.2) \times \text{Cycle time}
  \]

- What if we could double the CPU performance so the CPI becomes 0.5, but memory performance remains the same?

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

- The overall CPU time improves by just 1.2/0.7 = 1.7 times!

- Refer back to Amdahl’s Law from textbook page 51.
  - 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 the concrete example, 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 box 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 16 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 14 cycles.
  \[4 \times (1 + 1 + 1 + 1) = 14 \text{ clock cycles}\]

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 14 cycles.
  \[1 + 1 + 1 + 1 = 14 \text{ cycles}\]
- 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.
  \[1 + 15 + (4 \times 1) = 20 \text{ cycles}\]
- Our buses are still one byte wide here, so four cycles are needed to transfer data to the cache.
- 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.

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>Block size</th>
<th>Cache #1</th>
<th>Cache #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bytes</td>
<td>33%</td>
<td>48%</td>
</tr>
<tr>
<td>64-bytes</td>
<td>55%</td>
<td>41%</td>
</tr>
</tbody>
</table>

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

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>Block size</th>
<th>Cache #1</th>
<th>Cache #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bytes</td>
<td>55%</td>
<td>41%</td>
</tr>
<tr>
<td>64-bytes</td>
<td>55%</td>
<td>41%</td>
</tr>
</tbody>
</table>

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

recall: AMAT = Hit time × (Miss rate × 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.

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

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

\[
\text{CPU time} = (\text{CPU execution cycles} + \text{Memory stall cycles}) \times \text{Cycle time}
\]
- The organization of a memory system affects its performance.
  - The cache size, block size, and associativity affect the miss rate.
  - We can optimize the main memory to help reduce miss penalties. For
    example, interleaved memory supports pipelined data access.