Today we’ll finish up with caches; we’ll 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 see how to mitigate cache misses through pre-fetching.
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?

- Previous lectures answered the first 3. Today, 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>110</td>
<td>1</td>
<td>11010</td>
<td>42803</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 [but inconsistent]

\[
\text{Mem[1101 0110]} = 21763
\]

<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>
</tbody>
</table>

4/8/13  Cache performance 3
A write-through cache solves the inconsistency problem by forcing all writes to update both the cache and the main memory.

Mem[1101 0110] = 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>11010</td>
<td>21763</td>
</tr>
</tbody>
</table>

- 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.

```plaintext
Mem[1101 0110] = 21763
```

<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>42803</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></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.
- e.g. on a read from Mem[1000 1110], 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>
</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>21763</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>
</tr>
</thead>
<tbody>
<tr>
<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>
</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>21763</td>
</tr>
</tbody>
</table>

4/8/13 Cache performance
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” and written in background when memory is free.

- 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.
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.

When we update Mem[1101 0110], should we also load it into the cache?
Allocate on write

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

- If that data is needed again soon, it will be available in the cache.
- This is generally the baseline behavior or processors.

- What about the following?

```
for (int i = 0; i < LARGE; i++)
a[i] = i;
```
Non-temporal stores *(write-around/write-no-allocate)*

- For code where the stored values won’t get used in the near future, like:

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

- There is no point in putting these values in the cache.

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

  - Some modern processors with write-allocate caches provide special store instructions called *non-temporal stores* that do this.

<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>00010</td>
<td>123456</td>
<td>1101 0110</td>
<td>21763</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

4/8/13 Cache performance
Real Designs

Cache performance
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/DATA or CODE/data

- **Cache Hierarchies:**
  - Trade-off between access time & hit rate
    - L1 cache can focus on fast access time (with okay hit rate)
    - L2 cache can focus on good hit rate (with okay access time)
  - Such hierarchical design is another “big idea”
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.
  - We will do some cache simulations on the MP’s.
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.
- This graph 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.
- This graph 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.
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.
  - But if blocks are too large, there will be fewer blocks available, and more potential misses due to conflicts.
What happens on a cache miss?

- Can’t do write back (into register file) until data is fetched.
  - Easiest thing to do is stall immediately.
    - Sub-optimal if data isn’t used right away.

- Optimization: Non-blocking cache.
  - Remember miss and which register it should write into & continue
  - Stalls when:
    - Data is needed
    - Or too many misses outstanding

- Exploit by “hoist”ing loads up from their uses, but...
  - Uses up a register
  - For potentially many cycles (~100 to memory)
  - Might be guessing what will be accessed.

```
lw  $t0, 64($a0)
...
...
add  $v0, $t0, $t1
```
Software Prefetching

- Most modern architectures provide special software prefetch instructions
  - They look like loads w/o destination registers
    - e.g., on SPIM, \texttt{lw \$0, 64($a0)} \# write to the zero register.
  - These are hints to the processor:
    - “I think I might use cache block containing this address”
    - Hardware will try to move the block into the cache.
    - But, hardware can ignore (if busy)

- Useful for fetching data ahead of use:

```c
for (int i = 0 ; i < LARGE ; i ++) {
    prefetch A[i+16];  // prefetch 16 iterations ahead.
    computation A[i];
}
```
Prefetching, cont.

- Remember this graph?

Actual Data from remsun2.ews.uiuc.edu

```c
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}
```

Intel Core 2 Duo

Cache performance
Hardware Stream Prefetching

- Inner loop has very simple access pattern.
  - A, A+4, A+8, A+12, ...
  - What is called a stream

- We can easily build hardware to recognize streams
- If we get a pair of sequential misses (blocks X, X+1), predict a stream.
  - Fetch the next two blocks (X+2, X+3)

- Continue fetching the stream if the prefetch blocks accessed.
  - If X+2 is read/written, prefetch X+4 ...

- As confidence in stream increases, increase # of outstanding prefetches
  - If we get to X+8, have prefetches for X+9, X+10, X+11, X+12, X+13

- Can learn strides as well (X, X+16, X+32, ...) and (X, X-1, X-2, ...)

```c
int array[SIZE];
int A = 0;

for (int i = 0 ; i < 200000 ; ++ i) {
    for (int j = 0 ; j < SIZE ; ++ j) {
        A += array[j];
    }
}
```
PC-based HW Prefetching

- What about the following?

  ```c
  for (int i=0 ; i < LARGE ; i ++) {
    C[i] = A[i] + B[i];
  }
  ```

- 3 separate streams
  - Might confuse naïve prefetcher.

- Observation: A, B, and C accessed by different instructions.

- Learn a stream for each instruction

- Modern x86 chips do both stream, and PC-based stride prefetching in HW
So what do we need SW prefetching for?

- Non-stride accesses!
- Like linked data structures:
  - lists, arrays of pointers, etc.

- Consider:
  ```c
  element_t *A[SIZE];
  for (int i=0 ; i < SIZE ; i ++) {
    process(A[i]);
  }
  ```
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.

- Hardware prefetching handles most streams and strides.
  - We’ll talk later about 1 limitation.

- Software prefetching is useful for linked data structures
  - Must be added by programmer (or very smart compiler)

- Next time, we’ll look at cache-conscious programming.