Advance Caching
Way-associative cache

blocks sharing the same index are a “set”
Speeding up Memory

- \( ET = IC \times CPI \times CT \)
- \( CPI = \text{noMemCPI} \times \text{noMem}\% + \text{memCPI} \times \text{mem}\% \)
- \( \text{memCPI} = \text{hit}\% \times \text{hitTime} + \text{miss}\% \times \text{missTime} \)

- Miss times:
  - L1 -- 20-100s of cycles
  - L2 -- 100s of cycles

- How do we lower the miss rate?
Know Thy Enemy

• Misses happen for different reasons
• The three C’s (types of cache misses)
  • Compulsory: The program has never requested this data before. A miss is mostly unavoidable.
  • Conflict: The program has seen this data, but it was evicted by another piece of data that mapped to the same “set” (or cache line in a direct mapped cache)
  • Capacity: The program is actively using more data than the cache can hold.

• Different techniques target different C’s
Reducing Compulsory Misses

• Increase cache line size so the processor requests bigger chunks of memory.
• For a constant cache capacity, this reduces the number of lines.
• This only works if there is good spatial locality, otherwise you are bringing in data you don’t need.
• If you are reading a few bytes here and a few bytes there (i.e., no spatial locality) this will hurt performance
• But it will help in cases like this
  for(i = 0; i < 1000000; i++) {
    sum += data[i];
  }

  One miss per cache line worth of data
Reducing Compulsory Misses

• HW Prefetching

\[
\text{for}(i = 0; i < 1000000; i++) \{
\text{sum} += \text{data}[i];
\}
\]

• In this case, the processor could identify the pattern and proactively "prefetch" data program will ask for.
• Keep track of \text{delta} = \text{thisAddress} - \text{lastAddress}, it’s consistent, start fetching \text{thisAddress} + \text{delta}.
• Current machines do this alot... Prefetchers are as closely guarded as branch predictors.
• Learn lots more in 240A, if you’re interested.
Reducing Compulsory Misses

- Software prefetching
- Use register $zero!

```c
for(i = 0;i < 1000000; i++) {
    sum += data[i];
    if (i % 16 == 0) {
        "load data[i+16] into $zero"
    }
}
```

For exactly this reason, loads to $zero never fail (i.e., you can load from any address into $zero without fear)
Conflict Misses

- Conflict misses occur when the data we need was in the cache previously but got evicted.
- Evictions occur because:
  - Direct mapped: Another request mapped to the same cache line
  - Associative: Too many other requests mapped to the same cache line ($N + 1$ if $N$ is the associativity)

```c
while(1) {
    for(i = 0; i < 1024*1024; i+=4096) {
        sum += data[i];
    } // Assume a 4 KB Cache
}
Reducing Conflict Misses

- Conflict misses occur because too much data maps to the same “set”
  - Increase the number of sets (i.e., cache capacity)
  - Increase the size of the sets (i.e., the associativity)
- The compiler and OS can help here too
Colliding Threads and Data

- The stack and the heap tend to be aligned to large chunks of memory (maybe 128MB).
- Threads often run the same code in the same way.
- This means that thread stacks will end up occupying the same parts of the cache.
- Randomize the base of each thread's stack.
- Large data structures (e.g., arrays) are also often aligned. Randomizing malloc() can help here.
Capacity Misses

• Capacity misses occur because the processor is trying to access too much data.
  • Working set: The data that is currently important to the program.
  • If the working set is bigger than the cache, you are going to miss frequently.
• Capacity misses are bit hard to measure
  • Easiest definition: non-compulsory miss rate in an equivalently-sized fully-associative cache.
  • Intuition: Take away the compulsory misses and the conflict misses, and what you have left are the capacity misses.
Reducing Capacity Misses

- Increase capacity!
  - More associative sets (i.e., use more index bits)
- Costs area and makes the cache slower.
- Cache hierarchies do this implicitly already:
  - if the working set “falls out” of the L1, you start using the L2.
  - Poof! you have a bigger, slower cache.
- In practice, you make the L1 as big as you can within your cycle time and the L2 and/or L3 as big as you can while keeping it on chip.
Reducing Capacity Misses: The compiler

- The key to capacity misses is the working set
- How a program performs operations has a large impact on its working set.
Reducing Capacity Misses: The compiler

- Tiling
  - We need to make several passes over a large array
  - Doing each pass in turn will “blow out” our cache
  - “blocking” or “tiling” the loops will prevent the blow out
  - Whether this is possible depends on the structure of the loop
- You can tile hierarchically, to fit into each level of the memory hierarchy.
A Simple Example

• Consider a direct mapped cache with 16 blocks, a block size of 16 bytes, and the application repeat the following memory access sequence:

• 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010
A Simple Example

• a direct mapped cache with 16 blocks, a block size of 16 bytes

• $16 = 2^4 : 4$ bits are used for the index

• $16 = 2^4 : 4$ bits are used for the byte offset

• The tag is $32 - (4 + 4) = 24$ bits

• For example: $0x80000010$
A Simple Example

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>800000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>800000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x80000000 miss: compulsory
0x80000008 hit!
0x80000010 miss: compulsory
0x80000018 hit!
0x80000010 miss: compulsory
0x30000010 miss: compulsory
0x80000000 hit!
0x80000008 hit!
0x80000010 hit!
0x80000010 miss: conflict
0x80000018 hit!
A Simple Example: Increased Cache line Size

• Consider a direct mapped cache with 8 blocks, a block size of 32 bytes, and the application repeat the following memory access sequence:

• 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010
A Simple Example

- A direct mapped cache with 8 blocks, a block size of 32 bytes
- $8 = 2^3$ : 3 bits are used for the index
- $32 = 2^5$ : 5 bits are used for the byte offset
- The tag is $32 - (3 + 5) = 24$ bits
- For example: $0x80000010$
  - $011100000000000000000000000000010000$
    - **tag**
    - **index**
    - **offset**
# A Simple Example

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>800000</td>
</tr>
</tbody>
</table>

0x80000000 miss: compulsory
0x80000008 hit!
0x80000010 hit!
0x80000018 hit!

0x30000010 miss: compulsory
0x80000000 miss: conflict
0x80000008 hit!
0x80000010 hit!
0x80000018 hit!
A Simple Example: Increased Associativity

• Consider a 2-way set associative cache with 8 blocks, a block size of 32 bytes, and the application repeat the following memory access sequence:

• 0x80000000, 0x80000008, 0x80000010, 0x80000018, 0x30000010
A Simple Example

• a 2-way set-associative cache with 8 blocks, a block size of 32 bytes
• The cache has $8/2 = 4$ sets: 2 bits are used for the index
• $32 = 2^5 : 5$ bits are used for the byte offset
• The tag is $32 - (2 + 5) = 25$ bits
• For example: $0x80000010 =$
  • 0111000000000000000000000000001
  • 0000 tag
  • index offset
A Simple Example

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>600000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x80000000 miss: compulsory
0x80000008 hit!
0x80000010 hit!
0x80000018 hit!
0x30000010 miss: compulsory
0x80000000 hit!
0x80000008 hit!
0x80000010 hit!
0x80000018 hit!
End for Today
Increasing Locality in the Compiler or Application

• Live Demo... The Return!
Capacity Misses in Action

• Live Demo... The return! Part Deux!
Cache optimization in the real world: Core 2 duo vs AMD Opteron (via simulation)

(From Mark Hill’s Spec Data)

- .00346 miss rate
- .00366 miss rate
- Spec00
- Spec00
- Intel Core 2 Duo
- AMD Opteron

Intel gets the same performance for less capacity because they have better SRAM Technology: they can build an 8-way associative L1. AMD seems not to be able to.