So Far....

- Fully associative cache
  - Memory block can be stored in any cache block
- Write-through cache
  - Write (store) changes both cache and main memory right away
  - Reads only require getting block on cache miss
- Write-back cache
  - Write changes only cache
  - Read causes write of dirty block to memory on a replace
- Reads easy to make fast, writes harder
  - Read data from cache in parallel with checking address against tag of cache block
  - Write must verify address against tag before update

Write Buffers for Write-Through Caches

Q. Why a write buffer?
A. So CPU doesn’t stall

Q. Why a buffer, why not just one register?
A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?
A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Example: Alpha 21064

How much do stalls slow a machine?

- Suppose that on pipelined MIPS, each instruction takes, on average, 2 clock cycles, not counting cache faults/misses
- Suppose, on average, there are 1.33 memory references per instruction; memory access time is 50 cycles, and the miss rate is 2%
- Then each instruction takes, on average:
  
  \[
  2 + (0 \times .98) + (1.33 \times .02 \times 50) = 3.33 \text{ clock cycles}
  \]
Example: Apple iMac G5

<table>
<thead>
<tr>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
</tr>
</tbody>
</table>

Latency

- Cycles, Time:
  - 1, 0.6 ns
  - 3, 1.9 ns
  - 11, 6.9 ns
  - 88, 55 ns
  - 10^7, 12 ms

Goal: Illusion of large, fast, cheap memory
Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

Cache miss terminology

- Sometimes cache misses are inevitable:
  - Compulsory miss: The first time a block is used, need to bring it into cache
  - Capacity miss: If need to use more blocks at once than can fit into cache, some will bounce in and out
  - Conflict miss: In direct mapped or set associative caches, there are certain combinations of addresses that cannot be in cache at the same time

Miss rate

- Fig. C.9 SPEC2000, LRU replacement

5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
  5. Giving Reads Priority over Writes
    - E.g., Read completes before earlier writes in write buffer

More Terminology

- 'write-allocate': Ensure block in cache before performing a write operation
- 'write-no-allocate': Don’t allocate block in cache if not already there
Another write buffer optimization

- Write buffer mechanics, with merging
  - An entry may contain multiple words (maybe even a whole cache block)
  - If there’s an empty entry, the data and address are written to the buffer, and the CPU is done with the write
  - If buffer contains other modified blocks, check to see if new address matches one already in the buffer
  - If so, combine the new data with that entry
  - If buffer full and no address match, cache and CPU wait for an empty entry to appear (meaning some entry has been written to main memory)
  - Merging improves memory efficiency, since multi-word writes usually faster than one word at a time

Don’t wait for whole block on cache miss

- Two ways to do this – suppose need the 10th word in a block:
  - Early restart
    - Access the required word as soon as it is fetched, instead of waiting for the whole block
  - Critical word first
    - Start the fetch with word 10, and fill in the first few later

Use a nonblocking cache

- With this optimization, the cache doesn’t stop for a miss, but continues to process later requests if possible, even though an earlier one is not yet fulfilled
  - Introduces significant complexity into cache architecture – have to allow multiple outstanding cache requests (maybe even multiple misses)
  - but this is what’s done in modern processors

So Far (cont.)

- Reducing memory stalls
  - Reduce miss penalty, miss rate, cache hit time
- Reducing miss penalty
  - Give priority to read over write misses
  - Don’t wait for the whole block
  - Use a non-blocking cache

Multi-level cache

- For example, if cache takes 1 clock cycle, and memory takes 50, might be a good idea to add a larger (but necessarily slower) secondary cache in between, perhaps capable of 10 clock cycle access
- Complicates performance analysis (see H&P), but 2nd level cache captures many of 1st level cache misses, lowering effective miss penalty
  - and 3rd level cache has same benefits for 2nd level cache
- Most modern machines have separate 1st level instruction and data caches, shared 2nd level cache
  - and off processor chip shared 3rd level cache

Victim caches

- To remember a cache block that has recently been replaced (evicted)
  - Use a small, fully associative cache between a cache and where it gets data from
  - Check the victim cache on a cache miss, before going to next lower-level memory
    - If found, swap victim block and cache block
  - Reduces conflict misses
Victim caches (cont.)

How to reduce the miss rate?

- Use larger blocks
- Use more associativity, to reduce conflict misses
- Victim cache
- Pseudo-associative caches (won’t talk about this)
- Prefetch (hardware controlled)
- Prefetch (compiler controlled)
- Compiler optimizations

Increasing block size

- Want the block size large so don’t have to stop so often to load blocks
- Want the block size small so that blocks load quickly

Increasing block size (cont.)

- So large block size may reduce miss rates, but ...
- Example:
  - Suppose that loading a block takes 80 cycles (overhead) plus 2 clock cycles for each 16 bytes
  - A block of size 64 bytes can be loaded in $80 + 2 \times 64/16 = 88$ cycles (miss penalty)
  - If the miss rate is 7%, then the average memory access time is $1 + 0.07 \times 88 = 7.16$ cycles

Memory Access Times

<table>
<thead>
<tr>
<th>Cache size</th>
<th>Miss penalty</th>
<th>4K</th>
<th>16K</th>
<th>64K</th>
<th>256K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block size</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>82</td>
<td>8.027</td>
<td>4.231</td>
<td>2.673</td>
<td>1.894</td>
</tr>
<tr>
<td>32</td>
<td>84</td>
<td>7.082</td>
<td>3.411</td>
<td>2.134</td>
<td>1.588</td>
</tr>
<tr>
<td>64</td>
<td>88</td>
<td>7.160</td>
<td>3.323</td>
<td>1.933</td>
<td>1.449</td>
</tr>
<tr>
<td>128</td>
<td>96</td>
<td>8.469</td>
<td>3.659</td>
<td>1.979</td>
<td>1.470</td>
</tr>
<tr>
<td>256</td>
<td>112</td>
<td>11.651</td>
<td>4.685</td>
<td>2.286</td>
<td>1.549</td>
</tr>
</tbody>
</table>

Higher associativity

- A direct-mapped cache of size $N$ has about the same miss rate as a 2-way set-associative cache of size $N/2$
  - 2:1 cache rule of thumb (seems to work up to 128KB caches)
- But associative cache is slower than direct-mapped, so the clock may need to run slower
- Example:
  - Suppose that the clock for 2-way cache needs to run at a factor of 1.1 times the clock for 1-way cache
  - The hit time increases with higher associativity
  - Then the average memory access time for 2-way is $1.10 \times \text{miss rate} \times 50$ (assuming that the miss penalty is 50)
### Memory access time

Fig. C.13 - SPEC92 on DECstation 5000

<table>
<thead>
<tr>
<th>Cache size (KB)</th>
<th>One-way</th>
<th>Two-way</th>
<th>Four-way</th>
<th>Eight-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.44</td>
<td>3.25</td>
<td>3.22</td>
<td>3.28</td>
</tr>
<tr>
<td>8</td>
<td>2.69</td>
<td>2.58</td>
<td>2.53</td>
<td>2.62</td>
</tr>
<tr>
<td>16</td>
<td>2.23</td>
<td>2.16</td>
<td>2.18</td>
<td>2.25</td>
</tr>
<tr>
<td>32</td>
<td>2.06</td>
<td>2.00</td>
<td>2.00</td>
<td>2.00</td>
</tr>
<tr>
<td>64</td>
<td>1.92</td>
<td>1.84</td>
<td>1.92</td>
<td>2.00</td>
</tr>
<tr>
<td>128</td>
<td>1.32</td>
<td>1.66</td>
<td>1.74</td>
<td>1.82</td>
</tr>
<tr>
<td>256</td>
<td>1.20</td>
<td>1.55</td>
<td>1.59</td>
<td>1.66</td>
</tr>
</tbody>
</table>

- If cache big enough, slowdown in access time hurts mem performance

### Pseudo-associative cache

- Uses the technique of chaining, with a series of cache locations to check if the block is not found in the first location
  - E.g., invert most significant bit of index part of address (as if it were a set associative cache)
- The idea:
  - Check the direct mapped address
  - Until the block is found or the chain of addresses ends, check the next alternate address
  - If the block has not been found, bring it in from memory
- Three different delays generated, depending on which step succeeds

---

### How to reduce the cache miss rate?

- Use larger blocks
- Use more associativity, to reduce conflict misses
- Victim cache
- Pseudo-associative caches (won’t talk about this)
- Prefetch (hardware controlled)
- Prefetch (compiler controlled)
- Compiler optimizations

### Hardware prefetch

- Idea: If read page k of a book, the next page read is most likely page k+1
- So, when a block is read from memory, read the next block too
  - Maybe into a separate buffer that is accessed on a cache miss before going to memory
- Advantage:
  - If access blocks sequentially, will need to fetch only half as often from memory
- Disadvantages:
  - More data to move
  - May fill the cache with useless blocks
  - May compete with demand misses for memory bandwidth

### Compiler-controlled prefetch

- Idea: The compiler has a better idea than the hardware does of when blocks are being used sequentially
- Want the prefetch to be nonblocking:
  - Don’t slow the pipeline waiting for it
- Usually want the prefetch to fail quietly:
  - If ask for an illegal block (one that generates a page fault or protection exception), don’t generate an exception; just continue as if the fetch wasn’t requested
  - Called a non-binding cache prefetch

### Reducing the time for cache hits

- K.I.S.S.
- Use virtual addresses rather than physical addresses in the cache.
- Pipeline cache accesses
- Trace caches
K.I.S.S.

- Cache should be small enough to fit on the processor chip
- Direct mapped is faster than associative, especially on read
  - Overlap tag check with transmitting data
- For current processors, relatively small L1 caches to keep fast clock cycle time, hide L1 misses with dynamic scheduling, and use L2 and L3 caches to avoid main memory accesses

Use virtual addresses

- Each process has its own address space, and no addresses outside that space can be accessed
- To keep address length small, each user addresses by offsets relative to some physical address in memory (pages)
- For example:

<table>
<thead>
<tr>
<th>Physical address</th>
<th>Virtual address</th>
</tr>
</thead>
<tbody>
<tr>
<td>5400</td>
<td>00</td>
</tr>
<tr>
<td>5412</td>
<td>12</td>
</tr>
<tr>
<td>5500</td>
<td>100</td>
</tr>
</tbody>
</table>

Virtual addresses (cont.)

- Since instructions use virtual addresses, use them for index and tag in cache, to save the time of translating to physical address space (the subject of a later part of this unit)
- Note that it is important to flush the cache and set all blocks invalid when switch to a new user in the OS (a context switch), since the same virtual address then may refer to a different physical address
  - Or use the process/user ID as part of the tag in cache
- Aliases are another problem
  - When two different virtual addresses map to the same physical address – can get 2 copies in cache
    - What happens when one copy is modified?

Pipelined cache access

- Latency to first level cache is more than one cycle
  - We’ve already seen this in Unit 3
- Benefit is fast cycle time
- Penalty is slower hits
  - Also more clock cycles between a load and the use of the data (maybe more pipeline stalls)

Trace cache

- Find a dynamic sequence of instructions to load into a cache block, including taken branches
  - Instead of statically, from how the instructions are laid out in memory
  - Branch prediction needed for loading cache
- One penalty is complicated address mapping, since addresses not always aligned to cache block size
  - Can also end up storing same instructions multiple times
- Benefit is only caching instructions that will actually be used (if branch prediction is right), not all instructions that happen to be in the same cache block

Compiler optimizations to reduce cache miss rate
Four compiler techniques

- 3 techniques to improve cache locality:
  - Merging arrays
  - Loop interchange
  - Loop fusion
  - Loop blocking / tiling

Technique 1: merging arrays

- Suppose have two arrays:
  ```c
  int val[size];
  int key[size];
  ```
  and usually use both of them together

Merging arrays (cont.)

This is how they would be stored if cache blocksize is 64 words:

```
val[0] val[64] - -
val[1] val[65] - -
val[2] val[66] - -
val[3] val[67] - -
. . . .
val[size-1] - - -
key[0] - - -
key[1] - - -
key[2] - - -
key[3] - - -
```

Merging arrays (cont.)

Means that at least 2 blocks must be in cache to begin using the arrays.

```
val[0] val[64] - -
val[1] val[65] - -
val[2] val[66] - -
val[3] val[67] - -
. . . .
val[size-1] - - -
key[0] - - -
key[1] - - -
key[2] - - -
key[3] - - -
```

Merging arrays (cont.)

More efficient, especially if more than two arrays are coupled this way, to store them together.

```
val[0] val[32] val[64] -
key[0] key[32] - -
val[1] val[33] - -
key[1] key[33] - -
. . . .
```
Loop Interchange (cont.)

Notice that accesses are by columns, so the elements are spaced 1000 words apart.

Blocks are bouncing in and out of cache.

\[
\begin{align*}
&\text{for } i=0, 1, \ldots, 999 \\
&\text{for } j=0, 1, \ldots, 999 \\
&x[i][j] = 2 \times x[i][j]; \\
&\text{end for;}
\end{align*}
\]

No data dependences across loop iterations

Loop Interchange (cont.)

First color the loops:

\[
\begin{align*}
&\text{for } i=0, 1, \ldots, 999 \\
&\text{for } j=0, 1, \ldots, 999 \\
&x[i][j] = 2 \times x[i][j]; \\
&\text{end for;}
\end{align*}
\]

With new ordering, can exploit spatial locality to use every element in a cache block before needing another block!

Technique 3: loop fusion

Example:

\[
\begin{align*}
&\text{for } i=0, 1, \ldots, 999 \\
&\text{for } j=0, 1, \ldots, 999 \\
&x[i][j] = 2 \times x[i][j]; \\
&\text{end for;}
\end{align*}
\]

\[
\begin{align*}
&\text{for } i=0, 1, \ldots, 999 \\
&\text{for } j=0, 1, \ldots, 999 \\
&y[i][j] = x[i][j] \times a[i][j]; \\
&\text{end for;}
\end{align*}
\]

Loop fusion (cont.)

Note that the loop control is the same for both sets of loops.
**Loop fusion (cont.)**

And note that the array $x$ is used in each, so probably needs to be loaded into cache twice, which wastes cycles.

```plaintext
x[i][j] = 2 * x[i][j];
```

For $i=0, 1, \ldots, 999$

```plaintext
End for;
```

```plaintext
For j=0, 1, \ldots, 999

```

```plaintext
y[i][j] = x[i][j] * a[i][j];
```

For $i=0, 1, \ldots, 999$

```plaintext
End for;
```

```plaintext
For j=0, 1, \ldots, 999

```

**Loop fusion (cont.)**

So combine, or **fuse**, the loops to improve efficiency.

```plaintext
x[i][j] = 2 * x[i][j];
```

```plaintext
y[i][j] = x[i][j] * a[i][j];
```

```plaintext
```

Need to avoid introducing new capacity misses with fused loop.

**Technique 4: loop blocking / tiling**

**Example:**

```plaintext
x[i][j] = y[j][i];
```

For $i=0, 1, \ldots, 999$

```plaintext
End for;
```

```plaintext
For j=0, 1, \ldots, 999

```

Notice spatial reuse for both $i$ & $j$ loops (for $x$ & $y$)

Loop interchange would exploit spatial reuse for either $x$ or $y$, depending on which loop is outermost.

**Loop blocking / tiling (cont.)**

Blocking / tiling breaks loops into strips, then interchanges to form blocks

```plaintext
x[i][j] = y[j][i];
```

```plaintext
```

```plaintext
```

End for;
```

```plaintext
```

End for;
```

End for;
```

```plaintext
```

End for;
```

End for;
```

```plaintext
```

```plaintext
```

```plaintext
```

```plaintext
```

End for;
```

End for;
```

```plaintext
```

```plaintext
```

End for;
```

End for;
```

```plaintext
```

```plaintext
```

End for;
```

End for;
```

```plaintext
```

```plaintext
```

**Multi-Level Inclusion…**

- If all data in level $n$ is also in level $n+1$
  - Each bigger part of the memory hierarchy contains all data (addresses) in smaller parts
  - Not always the same data because of delayed writeback

- Why useful?
  - I/O…

- May be problematic
  - If block sizes differ between levels