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

Technique 1: merging arrays

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

Merging arrays (cont.)

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

```
```

Merging arrays (cont.)

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

```
```
Merging arrays (cont.)

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

```
val[0]  key[0]  val[32]  key[32]  ...
val[1]  key[32]  val[33]  key[33]  ...
...  ...  ...  ...
```

Technique 2: interchanging loops

Example:

```
For j=0, 1, ..., 99
For i=0, 1, ..., 4999
x[i][j] = 2 * x[i][j];
End for;
End for;
```

Interchanging loops (cont.)

Notice that accesses are by columns, so the elements are spaced 100 words apart.

Blocks are bouncing in and out of cache.

```
For j=0, 1, ..., 99
For i=0, 1, ..., 4999
x[i][j] = 2 * x[i][j];
End for;
End for;
```

Interchanging loops (cont.)

First color the loops:
Interchanging loops (cont.)

Notice that the program has the same effect if the two loops are interchanged:

```
For i=0, 1, …, 4999
  For j=0, 1, …, 99
    x[i][j] = 2 * x[i][j];
  End for;
End for;
```

But with this ordering, use every element in a cache block before needing another block!

```
For i=0, 1, …, 4999
  For j=0, 1, …, 99
    x[i][j] = 2 * x[i][j];
  End for;
End for;
```

Technique 3: loop fusion

Example:

```
For j=0, 1, …, 99
  For i=0, 1, …, 4999
    x[i][j] = 2 * x[i][j];
  End for;
End for;
```

Loop fusion (cont.)

Note that the loop control is the same for both sets of loops.

```
For j=0, 1, …, 99
  For i=0, 1, …, 4999
    x[i][j] = 2 * x[i][j];
  End for;
End for;
```

```
For j=0, 1, …, 99
  For i=0, 1, …, 4999
    y[i][j] = x[i][j] * a[i][j];
  End for;
End for;
```
### 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.

- For $j = 0, 1, \ldots, 99$
  - For $i = 0, 1, \ldots, 4999$
    - $x[i][j] = 2 * x[i][j]$;
    - End for;
  - End for;

- For $j = 0, 1, \ldots, 99$
  - For $i = 0, 1, \ldots, 4999$
    - $y[i][j] = x[i][j] * a[i][j]$;
    - End for;
  - End for;

### 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...

### Main memory management

- Questions:
  - How big should main memory be?
  - How to handle reads and writes?
  - How to find something in main memory?
  - How to decide what to put in main memory?
  - If main memory is full, how to decide what to replace?
The scale of things

- Typically (as of 2000):
  - Registers: < 1 KB, access time .25 - .5 ns
  - Cache: < 8 MB, access time .5 - 25 ns
  - Main Memory: < 4 GB, access time 150 - 250 ns
  - Disk Storage: > 30 GB, access time 5,000,000 ns (5ms)

- Memory Technology: CMOS (Complementary Metal Oxide Semiconductor)
  - uses a combination of n- and p-doped semiconductor material to achieve low power dissipation.

Memory hardware

- **DRAM:** dynamic random access memory, typically used for main memory
  - one transistor per data bit
  - each bit must be refreshed periodically (e.g., every 8 milliseconds), so maybe 5% of time is spent in refresh
  - access time < cycle time
  - address sent in two halves so that fewer pins are needed on chip (row and column access)

Memory hardware (cont.)

- **SRAM:** static random access, typically used for cache memory
  - 4-6 transistors per data bit
  - no need for refresh
  - access time = cycle time
  - address sent all at once, for speed