Reducing the Miss Ratio

There are three kinds of cache misses.

- **Cold misses.** To have something in the cache, first it must be fetched. The initial fetch of anything is a miss.
  These are also called *compulsory* misses or *first-reference* misses.

- **Capacity misses.** Misses that occur due to the limited capacity of the cache. Even in a fully associative cache, the block would have been replaced before being referenced again.
  Also called dimensional misses.

- **Conflict misses.** Occur because a block may be replaced because too many blocks map to its set, even if the cache would otherwise have enough space to keep it till the next reference.
  Conflict misses occur only in set-associative or direct-mapped caches.
  The difference between capacity and conflict misses: in the latter, the sets have limited capacity, even if the cache does not

  For example … all blocks in the program map into the same set in a 2-way set-associative cache. 100 blocks are accessed. Cache contains 1024 lines.

We can pursue strategies to minimize each of these kinds of misses.

First, however, let’s see how important each class of misses is.

On pp. 424–425, H&P have a table that shows that—

- Most misses are _______ misses—at least 66%, and usually more than 90%.
• The next most common kind are ______ misses—as many as 33%, but usually less than 10% in a set-associative cache.

• The rarest kind are ______ misses—no more than about 1%, or even fewer in a small cache.

Thus, the most obvious way to diminish the miss ratio would be to

Larger caches

*Advantage:* Larger caches hold more

*Disadvantages:*

• Steals resources from other units (especially for on-chip caches).

• Diminishing returns: double size ≠ double performance

• Larger caches are slower to access.

Larger line size

*Idea: Exploit spatial locality*
Problems:

- It doesn’t make the cache any larger.
- Too large a line size leads to *cache pollution* from data that will never be used.

- It also increases miss penalty (have to bring more in).

Looking at the data from H&P (p. 427), we find that larger caches can accommodate larger block sizes without incurring more misses.

Explain why this might be.

**Higher associativity**

*Advantage:* Removes conflict misses.

*Disadvantages:*

- Diminishing returns—4-way set associative is almost equivalent to fully associative in most cases
Prefetching
Idea: Cache it \textit{before} you need it.

Implementation:

\begin{itemize}
  \item +1 prefetch: Fetch missing block, and next sequential block
    Works great for streams with high sequential locality, e.g., instruction caches.
    Uses unused memory bandwidth between misses.
    Can “hurt” if there isn’t enough leftover bandwidth.
  \item Other prefetch strategies:
    Strided prefetch: note memory is being accessed every \( n \) locations, so prefetch block \( +n \).
    Example of code that has this behavior:
    \begin{verbatim}
    for (i = 1; i < MAX; i += n)
      a[i] = b[i];
    \end{verbatim}
\end{itemize}

Compiler-directed prefetch
The compiler will compile special code when prefetching is indicated.

To do this we need a “nonbinding prefetch” instruction that

\begin{itemize}
  \item doesn’t cause a page fault,
  \item doesn’t change processor’s state, and
  \item doesn’t delay processor on a miss.
\end{itemize}
The compiler predicts which accesses will miss, and inserts prefetch instructions far enough ahead to prevent the disaster of a cache miss.

For \( j = 0; j < 100; j++ \)
for \( i = 0; i < 100; i++ \)
\[ x[i][j] = c \cdot x[i][j] \]

\[ \text{for } (j = 0; j < 100; j++) \]
\[ \text{for } (i = 0; i < 100; i++) \{
\text{prefetch}(x[i+k][j]);
\text{x}[i][j] = c \cdot x[i][j];
\}
\]

where \( k \) depends on the miss penalty and the time it takes to execute an iteration

This reduces compulsory misses for the original instructions (the compulsory misses simply move around, since the prefetch instructions still generate the misses).

**Compiler-directed layout of instructions & data**

Idea: If we tend to fetch A then B, have the compiler put A and B in the same block (spatial locality)

For I-caches:

- For most instructions, this happens normally
- Branches change the sequential access pattern
- Solution:
  Figure out how frequently every branch is taken and not taken.
  Form the control flow graph.
  Draw boxes around largest sequential runs of code (see example on next page).
  Find groups of instructions that tend to execute one after another.
  Rewrite program putting these instructions into sequential order.
What order should we write out the blocks in?

**Layout for data via merging**

We can also reorganize data to diminish the number of cache misses.

We do this by laying out arrays that are accessed together in the same array (i.e., interleaved in memory).

What if block size = two “int”s? Each access to \(a[i]\) or \(b[i]\) brings in \(a[i+1]\) and \(b[i+1]\), which are never used.

How shall we change the layout to get better performance?
Now every access to a missing block brings in useful data as well.

Thus, we’ve enhanced the spatial locality of the code.