Direct Mapped Caches

Here is a series of address references given as addresses: 2, 11, 0, 73, 21, 11, 64, 27, 72, 28, 3, 28, 32, 33, 34, 35. Assuming a direct-mapped cache with 16 four-byte blocks that is initially empty, label each reference in the list as a hit or a miss and show the final contents of the cache.

<table>
<thead>
<tr>
<th>Index</th>
<th>Cache Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Set Associative Caches

A set-associative cache is a compromise between a direct-mapped cache and a fully-associative cache where each memory location maps to a set of cache locations. So, an N-way set associative cache has N cache locations in each set. If the address has m bits and the cache has $2^s$ sets with blocks of $2^o$ bytes, then the address splits up as follows:

\[
\begin{align*}
\text{Tag} &= m - s - o \\
\text{Index} &= s \\
\text{Block Offset} &= o
\end{align*}
\]

When a given set fills up, temporal locality tells us that the data which has not been accessed for the longest time is the least likely to be needed, so we use the replacement policy called least recently used, or LRU, that is, we assume the data we should replace is the least recently used data.

Problems

1. Why don’t we split addresses as follows?

\[
\begin{align*}
\text{Index} &= s \\
\text{Tag} &= m - s - o \\
\text{Block Offset} &= o
\end{align*}
\]
2. The L1 data cache of the AMD Barcelona is a 64KB, 2-way set-associative cache with 64-byte blocks. The Barcelona supports 40-bit physical addresses (i.e., can address 1TB of memory).

Compute the number of sets and the size of the tag, index, and block offset fields.

3. The L2 data cache of the Intel Core 2 Duo is a 2MB, 8-way set-associative cache with 64-byte blocks. The Core 2 Duo supports 36-bit physical addresses (i.e., can address 64GB of memory).

Compute the number of sets and the size of the tag, index, and block offset fields.

4. For the Barcelona, identify two addresses that map to the same set but are different blocks.
5. (a) **Cache 1:** Given a direct-mapped cache with 2 blocks of 2 bytes each, where the address is broken as follows:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

If the cache was initially empty and the addresses below were accessed in that order, which of the following would be cache misses? 0110 0010 0110 1100 0111 1001 0110 1100 1101

(b) **Cache 2:** Given a fully associative cache (using a least-recently-used replacement policy) with 3 blocks of 2 bytes each, where the address is broken as shown below.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Offset</th>
</tr>
</thead>
</table>

If the cache was initially empty and the addresses below were accessed in that order, which of the following would be cache misses? 0110 1101 0111 1001 1110 1100 0110 1111 1101

6. Give an access sequence for which Cache 1 has strictly more hits than Cache 2, or argue that such a sequence cannot exist.
Multilevel Cache

Cache is helpful only if the needed data is available there. On a cache miss, the processor still needs to access memory to retrieve the data. Each memory access is slow and results in high miss penalty. There are two ways to lower the amount of time lost on cache misses:

1. Decrease the **miss rate**. This implies building larger caches. But as the size of the cache increases, so does the average time to access it.

2. Decrease the **miss penalty**. We can’t just make memory faster, but we can put another, larger layer of cache on top of the slow memory. Even a very large cache is much faster than memory. This concept, called multilevel cache, is illustrated below.

![Figure 1. Concept of a multilevel cache](image)

On each load and store, the processor needs to access data in memory. First, it checks Level 1 cache (L1). If the data is found, it is read or written there. On L1 cache miss, the processor accesses Level 2 cache (L2) instead of memory. Since L2 is large, it is very likely that the data can be found there. For L2 cache, read/write time is longer than L1 but much shorter than memory. Thus the overall memory access time is lower.

With this cache hierarchy, we achieve good balance between size and speed:

1. By keeping the size of L1 cache small, we retain the high speed memory access to keep the CPU busy.
2. Enlarging the overall cache size by having large L2 cache provides a good backup for L1 cache misses without slowing down the response time on L1 cache hits.

Thus we can avoid the worst case: memory access on a cache miss.

**Can we add more levels of cache to achieve even better performance?**

The optimum number of levels of cache is determined by the trade-off between cache size and access latency. In practice, 3 levels is pretty common in modern designs with L1=32-64KB, L2=256KB-2MB, and L3=2MB-32MB.