### Homework 3 Solutions

B.2

a.

<table>
<thead>
<tr>
<th>Cache block</th>
<th>Set</th>
<th>Way</th>
<th>Possible memory blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M0, M1, M2, ..., M31</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>M0, M1, M2, ..., M31</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>M0, M1, M2, ..., M31</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
<td>M0, M1, M2, ..., M31</td>
</tr>
</tbody>
</table>

b.

<table>
<thead>
<tr>
<th>Cache block</th>
<th>Set</th>
<th>Way</th>
<th>Possible memory blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>M0, M2, ..., M30</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>M0, M2, ..., M30</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>M0, M2, ..., M30</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>M0, M2, ..., M30</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>M1, M3, ..., M31</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>M1, M3, ..., M31</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>M1, M3, ..., M31</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>M1, M3, ..., M31</td>
</tr>
</tbody>
</table>

Four-way set-associative cache organization.
B.5 A useful tool for solving this type of problem is to extract all of the available information from the problem description. It is possible that not all of the information will be necessary to solve the problem, but having it in summary form makes it easier to think about. Here is a summary:

- CPU: 1.1 GHz (0.909 ns equivalent), CPI of 1.35 (excludes memory accesses)
- Instruction mix: 75% nonmemory-access instructions, 20% loads, 10% stores
- Caches: Split L1 with no hit penalty (i.e., the access time is the time it takes to execute the load/store instruction
  - L1 I-cache: 2% miss rate, 32-byte blocks (requires 2 bus cycles to fill, miss penalty is 15 ns + 2 cycles
  - L1 D-cache: 5% miss rate, write-through (no write-allocate), 95% of all writes do not stall because of a write buffer, 16-byte blocks (requires 1 bus cycle to fill), miss penalty is 15 ns + 1 cycle
- L1/L2 bus: 128-bit, 266 MHz bus between the L1 and L2 caches
- L2 (unified) cache, 512 KB, write-back (write-allocate), 80% hit rate, 50% of replaced blocks are dirty (must go to main memory), 64-byte blocks (requires 4 bus cycles to fill), miss penalty is 60 ns + 7.52 ns = 67.52 ns
- Memory, 128 bits (16 bytes) wide, first access takes 60 ns, subsequent accesses take 1 cycle on 133 MHz, 128-bit bus

a. The average memory access time for instruction accesses:

- L1 (inst) miss time in L2: 15 ns access time plus two L2 cycles (two = 32 bytes in inst. cache line/16 bytes width of L2 bus) = 15 + 2 * 3.75 = 22.5 ns (3.75 is equivalent to one 266 MHz L2 cache cycle)
- L2 miss time in memory: 60 ns + plus four memory cycles (four = 64 bytes in L2 cache/16 bytes width of memory bus) = 60 + 4 * 7.5 = 90 ns (7.5 is equivalent to one 133 MHz memory bus cycle)
- Avg. memory access time for inst = avg. access time in L2 cache + avg. access time in memory + avg. access time for L2 write-back. =0.02 * 22.5 + 0.02 * (1 - 0.8) * 90 + 0.02 * (1 - 0.8) * 0.5 * 90 = 0.99 ns (1.09 CPU cycles).

b. The average memory access time for data reads:

Similar to the above formula with one difference: the data cache width is 16 bytes which takes one L2 bus cycles transfer (versus two for the inst. cache), so

- L1 (read) miss time in L2: 15 + 3.75 = 18.75 ns
- L2 miss time in memory: 90 ns
- Avg. memory access time for read = 0.02 * 18.75 + 0.02 * (1 - 0.8) * 90 + 0.02 * (1 - 0.8) * 0.5 * 90 = 0.92 ns (1.01 CPU cycles)
c. The average memory access time for data writes:
Assume that writes misses are not allocated in L1, hence, all writes use the write buffer. Also, assume the write buffer is as wide as the L1 data cache.
- L1 (write) time to L2: 15 + 3.75 = 18.75 ns
- L2 miss time in memory: 90 ns
- Avg. memory access time for data writes = 0.05 * 18.75 + 0.05 * (1 − 0.8) * 90 + 0.05 * (1 − 0.8) * 0.5 * 90 = 2.29 ns (2.52 CPU cycles)

d. What is the overall CPI, including memory accesses:
- Components: base CPI, Inst fetch CPI, read CPI or write CPI, inst fetch time is added to data read or write time (for load/store instructions).

\[
\text{CPI} = 1.35 + 1.09 + 0.2 \times 1.01 + 0.10 \times 2.52 = 2.84 \text{ cycles/inst.}
\]

B.8 a. Assume the number of cycles to execute the loop with all hits is \(c\). Assuming the misses are not overlapped in memory, then their effects will accumulate. So, the iteration would take

\[
t = c + 4 \times 100 \text{ cycles}
\]

b. If the cache line size, then every fourth iteration will miss elements of \(a, b, c,\) and \(d\). The rest of iterations will find the data in the cache. So, on the average, an iteration will cost

\[
t = (c + 4 \times 100 + c + c + c) / 4 = c + 100 \text{ cycles}
\]

c. Similar to the answer in part (b), every 16th iteration will miss elements of \(a, b,\) \(c,\) and \(d.\)

\[
t = (c + 4 \times 100 + 15 \times c) / 16 = c + 25 \text{ cycles}
\]

d. If the cache is direct-mapped and is of same size as the arrays \(a, b, c,\) and \(d,\) then the layout of the arrays will cause every array access to be a miss! That is because \(a_i, b_i, c_i,\) and \(d_i\) will map to the same cache line. Hence, every iteration will have 4 misses (3 read misses and a write miss). In addition, there is cost of a write-back for \(d_i\) which will take place in iterations 1 through 511. Therefore, the average number of cycles is

\[
t = c + 400 + 511 / 512 \times 100
\]
B.9 Construct a trace of the form addr1, addr2, addr3, addr1, addr2, addr3, addr1, addr2, addr3, ……., such that all the three addresses map to the same set in the two-way associative cache. Because of the LRU policy, every access will evict a block and the miss rate will be 100%.

If the addresses are set such that in the direct mapped cache addr1 maps to one block while add2 and addr3 map to another block, then all addr1 accesses will be hits, while all addr2/addr3 accesses will be all misses, yielding a 66% miss rate.

Example for 32-word cache: consider the trace 0, 16, 48, 0, 16, 48, …

When the cache is direct mapped address 0 will hit in set 0, while addresses 16 and 48 will keep bumping each other off set 16.

On the other hand, if the 32 word cache is organized as 16 set of two ways each. All three addresses (0, 16, 48) will map to set 0. Because of LRU, that stream will produce a 100% miss rate!

That behavior can happen in real code except that the miss rates would not be that high because of all the other hits to the other blocks of the cache.