Chapter 2/Appendix A sample problems

1. Let's compare the memory performance of a server versus desktop versus mobile device with reference to figure 2.1 on page 79 (or slide 3 of the chapter 2/appendix B notes). Use the fastest speeds for each layer's hit time as shown in figure 2.1. In order to obtain miss rates, use figure B.8 on page B-24 and assume all L1 caches are direct-mapped and L2 caches are 2-way set associative. For the L3 caches, assume for the desktop the miss rate is .002 and for the server the miss rate is .001. Assume DRAM never misses in all cases. Compute the effective access time formula for each machine.

   Answer:
   Mobile computer EAT = 1 ns + .037 * (5 ns + .012 * 50 ns) = 1.21 ns
   Desktop computer EAT = 1 ns + .037 * (3 ns + .012 * (10 ns + .002 * 50 ns)) = 1.12 ns
   Server EAT = 1 ns + .037 * (3 ns + .012 * (10 ns + .001 * 50 ns)) = 1.12 ns

   Notice that the server's performance is virtually the same as the desktop. The only difference between the two is the performance of the L3 cache, which has no tangible impact because the miss rate differences of .001 and .002 are not enough to cause any separation between the two computers' performances.

2. For decades, processor speed improved exponentially while memory speed improved linearly. The result was that the gulf between the two speeds diverged dramatically (memory miss penalties got worse and worse over the years). Starting around 2004, processor speedup began to level off. Does this mean that DRAM speeds are catching up? Does this mean that miss penalties are less important? Does this mean that we do not have to worry as much about miss rates?

   Answer: DRAM speeds are catching up but not significantly for two reasons. First, 20 years of seeing processor speed double while memory speed only increased linearly (or less) left the difference between the two at a very large rate. According to the graph on slide 2, the difference in performance (which is not the same as speed) is 1000-fold today. Typically what we see is L2 having an access time 2-5 times slower than L1 and DRAM being 25-50 times slower than L2 (or L3). So, while DRAM is slowly catching it, it will probably never reach the speed of processors. Miss penalties are somewhat less important not because of the speed difference but because miss rates have improved a huge amount. Part of this is due to cache improvements as we cover in chapter 2/appendix B, and part of this is due to having larger and more caches on the processor. We still have to worry about miss rates, but as just stated, miss rates have improved to 5% or less for L1 and 2% or less for L2, so continued improvement on miss penalties is having less of an impact than might be expected.

3. We have a direct-mapped cache of 1024 blocks, each block stores 16 words (a word is 32 bits in this case). We have an int array, a[65536], which is not currently stored in the data cache. Assuming i, c and n (which is 65536) are stored in registers and so they do not require any memory references, how many data cache misses will the following code create? Divide the misses into compulsory, capacity and conflict.

   ```
   for(j=0;j<4;j++)
   ```

```
for(i=0;i<n;i++)
        if(j==0||j==2) a[i]++;
        else a[i]*=2;
```

Answer:
Each block of the cache stores 16 array elements, so we will have a cache miss with every 16th access since every access is to a different array location, or 65536 / 16 = 4096 total data cache misses for each pass through the outer loop. As this is a direct-mapped cache, entries are discarded when another memory reference maps to the same block. Thus, because of the outer loop, the 4096 cache misses occur 4 times, or a total of 16384 cache misses. The first iteration of the outer loop causes compulsory misses because the array element that led to miss had not been loaded into the cache previously. Misses from the 2nd through 4th iteration are both capacity and conflict misses since the items were in the cache but thrown out because of both limited size of the cache and conflicts from the mapping function. Thus, the first 4096 misses are compulsory and the last 12288 misses are capacity/conflict misses.

4. Do problem B.2 on page B-60. Ignore the column "Way".

   Answer:
   a. Moving from a D-M to fully associative cache means that there is only 1 block, block 0. The block contains 8 sets.

Block	Set	Possible blocks
0	0-7	all blocks

   b. Moving from D-M to 4-way means that the 8 blocks are now divided into 2 blocks of 4 sets each.

Block	Set	Possible blocks
0	0-3	all even numbered blocks
1	0-3	all odd numbered blocks

5. The problem on slide 13 of the chapter 2/appendix B power point notes suggests that moving from a direct-mapped cache to a 2-way set associative cache requires lengthening the clock speed by 35%. This might be a bit exaggerated. Let's assume a 2-way set associative cache requires lengthening clock speed by 10%, 4-way by 20% and 8-way by 30%. Use the table in figure B.8, and assuming a base clock speed of .5 ns for a direct-mapped cache, and a miss penalty of 5 ns (an L2 cache), compute the access times for each type of cache (D-M, 2-W, 4-W, 8-W) for caches of sizes 4 KB, 16 KB and 64 KB.

   Answer:
   D-M 4K: .5 ns + .098 * 5 ns = .99 ns
   2-W 4K: .5 ns * 1.10 + .076 * 5 ns = .93 ns
   4-W 4K: .5 ns * 1.20 + .071 * 5 ns = .955 ns
   8-W 4K: .5 ns * 1.30 + .071 * 5 ns = 1.005 ns
   D-M 16K: .5 ns + .049 * 5 ns = .745 ns
   2-W 16K: .5 ns * 1.10 + .041 * 5 ns = .755 ns
   4-W 16K: .5 ns * 1.20 + .041 * 5 ns = .805 ns
   8-W 16K: .5 ns * 1.30 + .041 * 5 ns = .855 ns

D-M 64K: .5 ns + .037 * 5 ns = .685 ns
2-W 64K: .5 ns * 1.10 + .031 * 5 ns = .705 ns
4-W 64K: .5 ns * 1.20 + .030 * 5 ns = .75 ns
8-W 64K: .5 ns * 1.30 + .029 * 5 ns = .795

This shows that, unless the cache is very small, we would prefer a direct-mapped cache for an L1.

6. Provide an example of RISC-V code in which a cache miss's penalty is somewhat hidden because of out-of-order completion on a Tomasulo-style architecture as opposed to the RISC-V pipeline. In your example, explain how the pipeline would be impacted by the miss and at what point, if at all, the Tomasulo architecture is impacted by the miss.

Answer:

In the following code, assume all variables are floating point.

```
for(i=0;i<n;i++) {
        a[i] = b[i] + x;
        c[i] = c[i] * y;
        d[i] = c[i] + z;
        e[i] = d – w;
}
```

Let's assume that loading b[i] causes a cache miss. The pipeline would stall at that point and so would be unable to complete the first assignment statement let alone continue. In Tomasulo, the load of b[i] would be at a load/store functional unit and the addition of x would wait at an add functional unit, and while neither could continue, if there was another load/store functional unit, the next instruction (the multiply) could be issued and possibly begin execution. And even though the add functional unit would have the addition for b[i] and x, that instruction would wait at a reservation station so that, unless all of its reservation stations were full, that functional unit could also receive the issuing of the second addition and the subtraction. The second addition would have to wait for the multiplication but likely the multiplication would finish before the cache miss was fulfilled. The subtraction could not begin until both additions completed but at least some progress would be made. If that last assignment statement were not part of the loop, it is possible that the rest of the iteration could complete before the cache miss was handled!

7. Several cache optimizations revolve around either pipelined or parallel cache accesses: pipelined cache accesses, nonblocking caches, multibanked caches, critical word first caches, merging write buffers. Rank these in terms of how useful each is to support
    a. a Tomasulo-superscalar processor
    b. a MIPS-style single issue pipeline

Answer:

a. Here, we consider only data caches as a cache miss in an instruction cache will stall both forms of processors. The key to the Tomasulo-superscalar is to keep instructions flowing through it and so we do not want a data cache stall to badly impact its performance. Reducing miss rate would improve performance but none of these approaches impact miss rate. Therefore, we want to reduce load miss penalties as much as possible. We start with

the non-blocking cache. Without it, a stall means that no further data accesses can occur and so any instruction waiting on a load waits until the cache miss is handled. Next, critical word first/early restart will reduce the miss penalty so that the instruction waiting in a load/store unit, and as a result, instructions waiting on the datum in a reservation station, will wait less time. Multibanked caches and pipelined cache accesses can permit multiple loads/stores per cycle to help support a superscalar. Finally, merging the write buffer provides an improvement on write misses. Of all of these, this is the least important.

b. A stall freezes the entire pipeline so our goal is to reduce miss rates but as stated above, none of these techniques impact miss rate, so we use these approaches to reduce miss penalties. As stated above, critical word first will have the greatest impact on miss penalties but we need a non-blocking cache to implement this. Even though the non-blocking cache does not otherwise improve the single-issue pipeline, we need it, so these are the two most important. The multibanked cache is not necessary because any stall stalls the entire pipeline. A merging write buffer however can also reduce the miss penalty on a write. But in fact the most significant improvement is the pipelined cache access which would allow us to reduce hit time and speed up the processor. So I would rank them as pipelined cache access, non-blocking cache with critical word first, merging write buffer in that order. The multi-banked cache is less important.

8. A direct-mapped data cache has a 4-entry victim cache that stores the most recently discarded four blocks. On a cache miss, the victim cache is examined and if the item is found, the located block is *swapped* with the block currently in the D-M cache that it maps to (the block moved out of the D-M cache is not discarded, it is now stored in the victim cache). This provides a small degree of associativity. Assume the D-M cache has an access time of .5 ns. The hit time is 1 cycle but if the cache misses, it takes a further 1 cycle to access the victim cache and either 1 additional cycle for a swap (item found in the victim cache) or 10 cycles to go to the L2 cache. Also assume that the item is found in the victim cache 1/4 of the time. Compare the direct-mapped cache with victim cache to the same sized 2-way set associative cache which has a clock cycle time of .6 ns – which one should we use? Compare 4 KB, 16 KB and 64 KB caches. Use Figure B.8 on page B-24 for miss rates.

   Answer:
   Miss rates are: 4 KB D-M: .098, 4 KB 2-way: .076, 16 KB D-M: .049, 16 KB 2-way: .041, 64 KB D-M: .037, 64 KB 2-way: .031. The hit time for the D-M cache is 1 cycle (.5 ns) and for the 2-way set associative cache 1 cycle (.6 ns) miss. On a miss, the D-M cache has either a 2 cycle penalty (1 to access the victim cache and 1 to swap) or 11 cycles (1 to access the victim cache, 10 to access the L2 cache). The 2-way set associative cache has a miss penalty of 10 cycles. The access times are:

   4 KB D-M: .5 + .098 * (.25 * 2 * .5 + .75 * 11 * .5) = .929 ns
   4 KB 2-way: .6 + .076 * 10 * .6 = 1.056 ns
   16 KB D-M: .5 + .049 * (.25 * 2 * .5 + .75 * 11 * .5) =.714 ns
   16 KB 2-way: .6 + .041 * 10 * .6 = .846 ns
   64 KB D-M: .5 + .037 * (.25 * 2 * .5 + .75 * 11 * .5) = .662 ns
   64 KB 2-way: .6 + .031 * 10 * .6 = .786 ns

   Notice that the D-M + victim cache is better than the 2-way cache in all cases.

9. In way prediction, a 2-way set associative cache adds a predictor bit for each block, used to predict whether the request for the given is for set 0 or set 1. When a request comes in for a given block, the predictor bit is used to indicate which tag to look at, that of the given block's set 0 or set 1. If there is a hit, we are done and because the cache is not comparing tags in parallel, the cache's hit time can be the same as a direct-mapped cache. If the tag does not match the predicted set's tag, then the other set's tag is compared causing a 1 cycle penalty. On a match, the predictor bit is flipped to the other set and the item is returned with a 1 cycle penalty. On a second miss, it is a true cache miss and the item is fetched from lower in the memory hierarchy and brought into the cache. Let's examine how way prediction might improve performance over a direct-mapped cache and a standard 2-way set associative cache, and then consider how a 4-way set associative cache might implement way prediction. Assume the 1-bit prediction is 80% accurate. The cache access time (1 cycle) is .5 ns for the direct-mapped cache and the 2-way set associative cache with way prediction, and .6 for a standard 2-way set associative cache. The miss penalty is 5 ns.
    a. Compare a direct-mapped cache, 2-way set associative cache and 2-way set associative with way prediction where the cache size is 4 KB. Repeat for cache sizes of 32 KB and 128 KB.
    b. If we want to use way prediction for a 4-way set associative cache, how would you suggest it be implemented?

Answer:

Before we begin, figure B.8 will tell us the miss rates for the direct-mapped and 2-way set associative caches, but what about the way prediction cache? Because, on first access, it is acting like a direct-mapped cache, the miss rate will be the same as the D-M cache. On a miss, check the other set. Now the miss rate is that of the 2-way set associative cache. That second access will take place 20% of the time.

a. 4 KB:
    D-M: .5 ns + .098 * 5 ns = .99 ns
    2-way: .6 ns + .076 * 5 ns = .98 ns
    Way prediction: .5 ns +.20 * .5 ns + .076 * 5 ns = .98 ns

32 KB:
    D-M: .5 ns + .042 * 5 ns = .71 ns
    2-way: .6 ns + .038 * 5 ns = .79 ns
    Way prediction: .5 ns + .20 * .5 + .038 * 5 ns = .79 ns

128KB:
    D-M: .5 ns + .021 * 5 ns = .61 ns
    2-way: .6 ns + .019 * 5 ns = .70 ns
    Way prediction: .5 ns + .20 * .5 + .019 * 5 ns = .70 ns

Note that the 2-way and way prediction caches are identical in performance. Why? Because the miss-prediction, 20%, when multiplied by the D-M's hit time (.5 ns) gives us .1 ns, the exact difference in speed between the D-M and 2-way caches. Since the way prediction cache's miss rate is the same as the 2-way set associative cache's miss rate, we get identical performances. Aside from the smallest cache size (4 KB), the D-M outperforms both because the .1 ns is more significant than the miss rates as the D-M cache increases in size.

b. If the prediction fails on a 4-way set associative cache, where do we look next? We either need to utilize all the extra hardware to do a parallel tag check, as we would without any prediction, or we need additional prediction mechanisms, for instance by having a second and third prediction should the first fail. If we continue to make predictions, the penalty grows 1 cycle per missed prediction (for instance, if we are incorrect on the first prediction and correct on the second, it takes 2 cycles to access the cache, but if we are wrong on the first two predictions and right on the third, it takes 3 cycles). Whether we attempt more predictions or default back to the parallel tag checks, the cache requires more hardware than the 2-way prediction cache and in both cases, will take more time to fulfill that second (or third or fourth) attempt. Because of the complexity and slowdown, we probably would not use way prediction for anything but the 2-way cache.