

CSE141 Problem Set #4 Solutions

March 5, 2002

1 Simple Caches

For this first problem, we have a 32 Byte cache with a line length of 8 bytes. This means that we have a total of 4 cache blocks (cache lines) to work with. The following is the sequence of memory references we are considering: 16, 4, 40, 32, 0, 8, 44, 16. Let us consider the binary representations of these numbers:

```
16  010000
4   000100
40  101000
32  100000
0   000000
8   001000
44  101100
16  010000
```

- **Direct mapped cache**

A direct mapped cache is synonymous with 1-way set associative cache. This means that each "set" of our cache can only accommodate a single cache block. Therefore, since we have a total of 4 cache blocks, we must also have 4 sets, and consequently need 2 bits to index into the cache. Counting from the right of one of our given address references, we know that bits 0, 1, and 2 are used for the offset, since our cache line size is 8 bytes. Bits 3 and 4 will then serve as the index for the cache. Please draw a single-column box with 4 rows to represent our cache, and label the four rows from top to bottom 00, 01, 10, and 11, respectively. Feel free to fill in the cache as you read the following:

- 16 indexes to 10, and so 10 gets bytes 16-23
- 4 indexes to 00, and so 00 gets bytes 0-7
- 40 indexes to 01, and so 01 gets bytes 40-47
- 32 indexes to 00, and so 00 gets bytes 32-39, kicking out 0-7
- 0 indexes to 00, and so 00 gets bytes 0-7, kicking out 32-39
- 8 indexes to 01, and so 01 gets bytes 8-15, kicking out 40-47

- 44 indexes to 01, and so 01 gets bytes 40-47, kicking out 8-15
- 16 indexes to 10, and we have a **cache hit**

The final state of the cache is as follows: 00 has bytes 0-7, 01 has bytes 40-47, and 10 has bytes 16-23. 11 is blank. We had a total of 1 cache hit.

- **2-way set associative cache with LRU**

For this cache, our line size is still 8 bytes, but instead of each set being able to accommodate just one cache block, it can accommodate two. Please draw a box with two columns and two rows, and label the top row 0, and the bottom row 1. Notice that although we still use bits 0, 1, and 2 for the offset, we now only need bit 3 to index into the cache. The execution goes as follows:

- 16 indexes to 0, and so the first block of 0 gets bytes 16-23
- 4 indexes to 0, and so the second block of 0 gets bytes 0-7
- 40 indexes to 1, and so the first block of 1 gets bytes 40-47
- 32 indexes to 0, and so the first block of 0 gets bytes 32-39, kicking out 16-23
- 0 indexes to 0, and we have a **cache hit** in the second block of 0
- 8 indexes to 1, and so the second block of 1 gets bytes 8-15
- 44 indexes to 1, and we have a **cache hit** in the first block of 1
- 16 indexes to 0, and so the first block of 0 gets bytes 16-23, kicking out 32-39

The final state of the cache is as follows: the two blocks of 0 have bytes 16-23 and 0-7, and the two blocks of 1 have bytes 40-47 and 8-15. We had a total of 2 cache hits.

- **Fully-associative cache with LRU**

For this cache, we have but a single set that can hold 4 cache blocks. We still use bits 0, 1, and 2 to determine the offset, and we no longer need bits for the index at all since there is only a single set to map to. Please draw a box with 4 columns and one row. The execution goes as follows:

- Block 1 gets bytes 16-23
- Block 2 gets bytes 0-7
- Block 3 gets bytes 40-47
- Block 4 gets bytes 32-39
- 0-7 is already in block 2, so we have a **cache hit**
- Block 1 gets bytes 8-15, kicking out 16-23
- 40-47 is already in block 3, so we have a **cache hit**
- Block 4 gets bytes 16-23, kicking out 32-39

The final state of the cache is as follows: the four blocks of the cache contain bytes 8-15, 0-7, 40-47, and 16-23, respectively. We had a total of 2 cache hits.

2 Higher associativity = better performance?

In this example, we see a situation where higher associativity in the cache actually has a negative impact on cache performance for a certain code segment. Recall the code segment in question:

```
for (i=0; i<10; i++)
    for (j=0; j<600; j++)
        sum += a[i]*b[j];
```

From the problem description, we are only concerned with the references to the `b[]` array. We also know that our cache is 2KB, or 2048 Bytes, and that our cache line size is 16 Bytes. To find the number of cache blocks we have to work with, we simply divide 2048 by 16:

$$\frac{2048}{16} = \frac{2^{11}}{2^4} = 2^7 = 128$$

- **Fully-associative cache with LRU**

If we simply look at what happens when execution begins, it is fairly easy to notice a pattern in how this scheme works. For the sake of simplicity, let us assume that the `b[]` array has a base address of 0. The first execution of the inner loop has a total of 600 iterations, covering 600 data elements of `b[]`, which are each 4 Bytes long. Assuming the cache is initially empty, the first data reference is to `b[0]`, or address 0. This cache miss will result in bytes 0-15 being brought into the first cache block. The next three iterations which reference `b[1]`, `b[2]`, and `b[3]`, or addresses 4, 8, and 12, respectively, will all hit due to our 16 Byte cache line size. The next four iterations will follow a similar pattern, with the reference to `b[4]` (address 16) causing a miss that brings in bytes 16-31, and the three subsequent iterations for `b[5]`, `b[6]`, and `b[7]` (addresses 20, 24, and 28) hitting in the very block that was just brought in.

Now we know that the entire `b[]` array cannot fit entirely into cache, since there are only 128 cache blocks, and each cache block can essentially handle 4 data elements each. This means that once we have populated all 128 cache blocks, we have 512 data elements of `b[]` in cache, with yet another 88 to reference for the remainder of the inner loop. As we continue to reference elements of `b[]`, we must begin using LRU to kick out previous cache entries. Let us consider the moment when our cache is full and contains data elements 0-511 of `b[]`. When we make the reference to `b[512]`, based on LRU we must kick out elements 0-3 which are sitting in the first block of the cache and replace them with elements 512-515. Just as before, we will hit the next three references, but when we reference 516, we will kick out elements 4-7 from the second cache block and replace them with 516-519. By the time we finish our initial inner loop of 600, and then begin our second execution of the inner loop for another 600 references, it is clear that `b[0]` and a good portion of the early elements of `b[]` are long gone out of the cache, replaced by elements of `b[]` in the 500's range. Furthermore, as we start back from `b[0]` on upward, it is clear that this pattern will continue, and that our program will essentially be kicking out of cache the very data it will find it needs in around 88 iterations. Therefore, we see that our fully-associative cache is doomed to have a maximum

hit rate of 75%, always missing with the first element of the cache block, and hitting with the next three.

- **Direct mapped cache** Things with the direct mapped cache work a little differently. The key observation to be made is that each memory address maps to one and only one location in the cache. If we take a look at some of the key values between data elements $b[0]$ - $b[599]$ which correspond to addresses 0 through 2396: (keep in mind that with a 16 Byte line size, there are 4 bits for the offset, and with 128 sets, there are 7 bits for the index)

$b[]$	address	binary address	cache set
$b[0]$	0	000000000000	0
$b[4]$	16	000000010000	1
$b[508]$	2024	011111110000	127
$b[511]$	2044	011111111100	127
$b[512]$	2048	100000000000	0
$b[596]$	2384	100101010000	21
$b[599]$	2396	100101011100	21

Let's imagine that we are starting program execution from an initially empty cache. Our cache has 128 sets, numbered 0 to 127. At the start, set 0 gets $b[0-3]$, set 1 gets $b[4-7]$, etc. This will continue until set 127 gets $b[508-511]$. At this moment, our cache is entirely populated. In addition, we have followed a similar pattern as the fully-associative cache by having one miss followed by three hits due to the fact that our cache line size is 16 Bytes. As we proceed with $b[512-515]$, we hash back to the 0 set, and kick out $b[0-3]$. This continues until we finally end the loop with $b[596-599]$ which hashes to set 21. We have essentially "overwritten" sets 0-21 with the latter part of the $b[]$ array. The observation to make at this point is that there are only 22 sets out of 128 that actually have more than one set of addresses mapping to them. In other words, as we start our next 600 iterations, rather than continuing to kick out sets 22, 23, 24, etc. as we would if we were using LRU, we start back at 0 again. We kick out sets 0-21, but then sets 22-127 have exactly what we want. Therefore, we can total up our misses as follows:

- For the first set of 600 iterations, we have to populate the entire cache, and so we essentially have the same 1 miss followed by 3 hits pattern as we saw in the fully-associative version of the cache. Therefore, we have 450/600 hits.
- For the remaining 9 sets of 600 iterations, for sets 0-21 we will have to follow the same pattern, kicking out the high end of the $b[]$ array and replacing it with the beginning of the $b[]$ array. Then, sets 22-127 are already in cache and give a 100% hit rate. Finally, for the last part of the $b[]$ array, we again have to kick out sets 0-21 and follow the same 1 miss 3 hit pattern for those iterations.

This is the total for one pass of 600 with a 75% hit rate, and 9 passes where we have 22 sets at 75% followed by 106 sets at 100% followed by another 22 sets at 75%. This is clearly better than the performance of the fully-associative cache that had an overall hit rate of 75%.

Total number of accesses = 6000

Total number of hits = $450 + 9[(22*3) + (106*4) + (22*3)] = 450 + 9(556) = 5454$ hits

Hit rate = $\frac{5454}{6000} = 90.9\%$

3 Addendum: Miss definitions

Some of you had asked about the definitions of compulsory, capacity, and conflict misses in section last week. First of all, compulsory misses are fairly clear cut. Regardless of the state of the cache, if this is the first time a program is referencing a particular location, this is classified as a compulsory miss. As far as categorizing the remaining misses as either conflict or capacity, it is first important to realize that one cannot really look at an individual miss out of context and determine whether or not to classify it as capacity or conflict. It depends on the caches in question as well as the memory locations being referenced. It is therefore better to look at the problem in terms of how the cache in question could benefit from associativity. If a fully associative cache of the same size reduces misses, then those original misses were conflict misses; however, if making the cache fully associative doesn't affect the misses, then we are clearly dealing with capacity misses. This follows from the general principle that higher associativity should help reduce conflict misses.