ECE 565 - HW2

Yi Mi

September 22, 2020

1 VM information

Hostname: vcm-12347.vm.duke.edu
Operating System: Ubuntu18 Server
Base memory: 2 GB
Processors: 2

2 Question 1

Organization:
- Total cache size: 512 bytes
- Cache blocks: 64 bytes
- 2-way set associative
- LRU replacement policy

Then we have:
- Offset bits = $log_2 64 = 6$
- Set = 512 / (64*2) = 4 => Set bits = $log_2 4 = 2$
- Tag bits = 20 - 6 - 2 = 12

Figure 1: Cache hit or miss

Address	Tag	Set	Offset	Hit or Miss
0xABCDE	101010111100	11	011110	M
0x14327	000101000011	00	100111	M
0xDF148	110111110001	01	001000	M
0x8F220	100011110010	00	100000	M
0xCDE4A	110011011110	01	001010	M
0x1432F	000101000011	00	101111	H
0x52C22	010100101100	00	100010	M
0xABCF2	101010111100	11	110010	H
0x92DA3	100100101101	10	100011	M
0xF125C	111100010010	01	011100	M

Figure 2: Final contents of the cache

Set	LRU	Way 1			Way 0		
		V	Tag	Data	V	Tag	Data
00	0	1	010100101100	0x52C22	1	000101000011	0x14327
01	1	1	110011011110	0XCDE4A	1	111100010010	0xF125C
10	1				1	100100101101	0x92DA3
11	1				1	101010111100	0xABCDE

3 Question 2

Average access time = hit time + (miss rate * miss penalty)
(a) AAT = 1 + 3% * (15 + 30% * 300) = 4.15 CPU cycles
(b) AAT = 1 + 10% * (15 + 5% * 300) = 4 CPU cycles

4 Question 3

(a)
The size of L1 data cache is 32768. The code is attached in the folder problem3/.

(b)
- Folder name: problem3/
- Optimization level: -O3
- Program name: test_bandwidth.c
- Compiled binary program: test_bandwidth
- Build script: build.sh

(c)
I used the full size of L1 cache which is 32768 byte as the size of my data structure, an array, and I initialized the array by assign array[i] = i. To stress bandwidth, I use a loop and traverse 10,000 times. I also avoid using dependent instruction. To achieve the write only pattern, I assign the each array element as a new number which is i+1. For 1:1 read-to-write pattern, I assign one array element to a temp variable and assign a new value to another element. For 2:1 read-to-write pattern, I read and add two array elements and assign a new value to another element. Since I run the loop three times in the main function and calculate time separately, I can just run bash ./build and ./test_bandwidth to get the results for all three patterns.

Figure 3: Three patterns bandwidth

Traffic	1st run (GB/s)	2rd run (GB/s)	3rd run (GB/s)	Average bandwidth (GB/s)
Write traffic only	5.107297	4.952819	5.073046	5.044387
1:1 read-to-write ratio	10.563411	10.615328	10.562850	10.580530
2:1 read-to-write ratio	15.840392	15.967150	15.805189	15.870910

I run three times and calculate the average bandwidth. From the table above we can see that the 2:1 read-to-write traffic has the highest bandwidth and write traffic only has the lowest bandwidth. The reason is there isn't any caching or locality on writes but for read there could be good spatial locality which increases the bandwidth.

(d)
I assign the size of the data structure to be the double of L3 cache size. Run ./test_bandwidth_large to get the results. The bandwidth for 1:1 read-to-write traffic and 2:1 read-to-write traffic are decreased for the reason that the fetch time increases with going down to the lower level caches.
- Folder name: problem3/
- Optimization level: -O3
- Program name: test_bandwidth_large.c
- Compiled binary program: test_bandwidth_large
- Build script: build.sh

Figure 4: Three patterns bandwidth with large data structure

Traffic	1st run (GB/s)	2rd run (GB/s)	3rd run (GB/s)	Average bandwidth (GB/s)
Write traffic only	7.425701	6.702367	7.201609	7.109892
1:1 read-to-write ratio	7.428872	6.595029	7.414685	7.146195
2:1 read-to-write ratio	7.445665	6.701571	7.336433	7.161223

5 Question 4

(a)
The code is attached in the folder problem4/. Run bash ./build.sh to compile and then run ./matrix_multiplication 1, ./matrix_multiplication 2, and ./matrix_multiplication 3 to control which nest ordering is used.
- Folder name: problem4/
- Optimization level: -O3
- Program name: matrix_multiplication.c
- Compiled binary program: matrix_multiplication
- Build script: build.sh

(b)
The misses per iteration:
- k innermost: 1 + (element sz / block sz)
- i innermost: 2
- j innermost: 2 * (element sz / block sz)

Figure 5: Run time of different ordering loop

Order	1st run (s)	2rd run (s)	3rd run (s)	Average time (s)
i-j-k	7.837859	7.870226	7.843664	7.850583
j-k-i	39.038134	38.657691	39.639016	39.111614
i-k-j	0.628926	0.639489	0.643481	0.637299

From the rank aspect, it matches my expectations that i-j-k time > j-k-i time > i-k-j time for the reason that j-innermost loop has both good spatial locality and good temporal locality. But from the number aspect, my expectation is that j-k-i time should be roughly twice as much as i-j-k time and now it is much higher than that.

(c)
As we see before, the L2 cache size is 1048576. I chose 64 as the sub-block size. The loop code is attached below and the complete code is in matrix_multiplication_tiling.c. I tiled each loop of the multiplication. After running bash ./build.sh, run ./matrix_multiplication_tiling to get the time.

- Folder name: problem4/
- Optimization level: -O3
- Program name: matrix_multiplication_tiling.c
- Compiled binary program: matrix_multiplication_tiling
- Build script: build.sh

```
for (i=0; i<N; i+=64) {
    for (j=0; j<N; j+=64) {
        for (k=0; k<N; k+=64) {
            for (ii=i; ii<i+64; ii++) {
                for (jj=j; jj<j+64; jj++){
                    sum = 0;
                    for (kk=k; kk<k+64; kk++){
                        sum+=A[ii][kk] * B[kk][jj];
                    }
                    C[ii][jj] = sum;
                }
            }
        }
    }
}
```

(d)
From the table we can see that the tiling version of i-j-k order is much faster than the non-tiling version since we make use of space locality. But it is still slower than the j-innermost loop since it has both B and C good spatial locality and A good temporal locality. When k is the innermost loop and the loop is tiled, the misses per iteration (1 + element sz / block sz) could increase but the misses per iteration for j-innermost loop condition (2 * element sz / block sz) should always be lower than that.

Figure 6: Run time of two versions of k-innermost loop

Order	1st run (s)	2rd run (s)	3rd run (s)	Average time (s)
i-j-k	7.837859	7.870226	7.843664	7.850583
i-j-k (Tiling version)	1.163529	1.181143	1.162107	1.168926