Homework 2

Yuanrui Zhang(yz545)

Problem 1

According to the problem description, we know that cache block size is 64B, so block offset is 6 bits. Total cache size is 512B, so we can calculate that there are 512B / 64B = 8 blocks. In view of 2-way set associative, the number of sets is 8 / 2 = 4, which can be represented by 2 bits. Note that the address space is 20 bits, therefore, we can use 20-6-2 = 12 bits as a tag.

Trace:

Address	**Tag bits**	**Index bits**	**Block offset bits**	**Result**
0xABCDE	101010111100	11	011110	Miss
0x14327	000101000011	00	100111	Miss
0xDF148	110111110001	01	001000	Miss
0x8F220	100011110010	00	100000	Miss
0xCDE4A	110011011110	01	001010	Miss
0x1432F	000101000011	00	101111	Hit
0x52C22	010100101100	00	100010	Miss
0xABCF2	101010111100	11	110010	Hit
0x92DA3	100100101101	10	100011	Miss
0xF125C	111100010010	01	011100	Miss

Final cache content:

Set #	**Way 0**			**Way 1**		
	Valid	**Tag**	**Data**	**Valid**	**Tag**	**Data**
0	1	000101000011	0x1432F	1	010100101100	0x52C22
1	1	111100010010	0xF125C	1	110011011110	0xCDE4A
2	1	100100101101	0x92DA3	0	\	\
3	1	101010111100	0xABCF2	0	\	\

Problem 2

a) AAT = 1 + 0.03*(15 + 0.3*300) = 4.15 cycles
b) AAT = 1 + 0.1*(15 + 0.05*300) = 4 cycles

Problem 3

(a) Platform: Linux VM. L1 cache size: 32KB.

(b) Experiment screenshot (as a sample):

```
yz545@vcm-15645:~/ECE565/Homework_repo/HW2/bandwidth_measurement$ ./bandwidth_gauger 4096 100000
**********Write traffic only**********
Time = 138.202329 ms
Bandwidth is = 23.710165 GB/s

**********1:1 read-to-write traffic**********
Time = 138.242488 ms
Bandwidth is = 47.406554 GB/s

**********2:1 read-to-write traffic**********
Time = 110.494308 ms
Bandwidth is = 88.967479 GB/s
```

(c) The number of elements in the following table is 4096, because I used an uint64 array to test bandwidth, 8B per element. 4096 elements can exactly fill L1 cache.

I stress bandwidth by introducing a command line argument: num_traversals.

To achieve different access patterns, I apply different array index calculations:

1. Write only:

```
for(int i = 0; i < num_traversals; i++) {
    for(int j = 0; j < num_elements; j++) {
        array[j] = 8;
    }
}
```

So that each access writes to corresponding array location once. No reads included.

2. Read:write = 1:1

```
for(int i = 0; i < num_traversals; i++) {
    for(int j = 0; j < num_elements; j++) {
        array[j] = array[j] + 8;
    }
}
```

So that each access reads first then update its value.

3. Read:write = 2:1

```
for(int i = 0; i < num_traversals; i++) {
    for(int j = 0; j < num_elements - 1; j++) {
        array[j] = array[j] + array[j + 1];
    }
    array[num_elements - 1] = array[num_elements - 1] + array[0];
}
```

So that in the inner loop, I read two values, sum them up and write back once.

Usage:
./bandwidth_gauger <num_elements> <num_traversals>

Results:
The following table records bandwidth based on different number of traversals and access patterns:

num_traversals / access patterns	100,000	1,000,000	10,000,000
Write only	22.865009 GB/s	23.691270 GB/s	23.442164 GB/s
Read:write = 1:1	47.199279 GB/s	46.871620 GB/s	46.123013 GB/s
Read:write = 2:1	88.788531 GB/s	88.132108 GB/s	87.767868 GB/s

Note that I don't include time in the table, but each testing time is long enough to guarantee stability. For the number of traversals 100,000, 1,000,000 and 10,000,000, their execution time is ~100ms, ~1000ms and ~10000ms respectively.

Discussion:
According to the results, read:write = 2:1 reaches highest bandwidth. As we increase the ratio of reads, the bandwidth increases accordingly. Therefore, I can reasonably speculate this L1 cache has write channels and read channels and more than 1 read channels. Isolating write and read is beneficial to parallelize, hence increase the performance.

(d) The largest cache size on my machine is 25344KB. So I can choose the number of elements in my array > 25344KB / 8B = 3244032 → I make it 3,500,000, 7,000,000 and 10,000,000 respectively to see the results. Note that, this time I keep the number of traversals fixed, 10.

num_elements / access patterns	3,500,000	7,000,000	10,000,000
Write only	11.847571 GB/s	10.609930 GB/s	10.222852 GB/s
Read:write = 1:1	25.079441 GB/s	22.189149 GB/s	22.044579 GB/s
Read:write = 2:1	35.027063 GB/s	31.123735 GB/s	30.892325 GB/s

Summary:
As I increase the number of elements in the array to surpass the volume the L3 cache, it is apparent that the throughput drops sharply compared to L1 cache's for all the access patterns. I also observe that, from 3,500,000 to 7,000,000, bandwidth continues decreasing; while further to 10,000,000, it stops.
They do match my expectation because in this case, CPU has to requests from main memory, which is definitely slower than from caches.

Problem 4

(a) Platform: Linux VM
Mode control (how to run my program)
./matrix <MODE>
If no mode arguments are provided, it will run first 3 modes (see below) one by one. Otherwise, only 4 modes are available:
Mode1: I-J-K;
Mode2: I-K-J;
Mode3: J-K-I;
Mode4: I-J-K with loop tiling
E.g., if you type, *./matrix 1*, it will run I-J-K pattern alone.

(b) I-J-K: 1.841612 s
I-K-J: 0.487556 s
J-K-I: 19.799538 s

These results match the note from lecture, but only qualitatively not quantitively. According to the class note, cache misses per inner loop iteration is:
I-J-K: 1.125
I-K-J: 0.25
J-K-I: 2
The statistics means that I-K-J is the better pattern, 4.5 times better than I-J-K and 8 times better than J-K-I, in terms of performance (here, execution time). Therefore, we can clearly see from (b) that the performance ranking matches theory, but the numerical relations don't. The discrepancy happens on J-K-I, which is practically 40 times worse than I-K-J. One possible reason is that because the locality is compromised, every time CPU fetches next element, it actually needs to request from lower level caches or even from main memory, which increases latency drastically. Another possible can be I am using -O3 optimization, which compiler can take great advantage of locality and can't do any optimization if no spacious locality can be found, like I-K-J pattern.

(c) L2 cache size: 1024KB.
Set such sub-block size as x. Calculate inequation:
$3x^2 * 8 <= 1024KB \rightarrow x <= 209$
Run my program: *./matrix 4*

(d) The tiled version of matrix multiplication runs 1.380306 s, while the normal one runs 1.822339 s.
The block size = 128 (calculation in part (c))
Tiled version is faster I-J-K and J-K-I patterns but still slower than I-K-J.

Discussion:
The tiled version does have performance improvement over plain I-J-K pattern because loop tiling helps improve cache reuse and make vectorization easier. However, I-K-J

pattern (with the best use of cache locality) still ranks the first among all versions because spacious locality dominates. Decreasing cache misses can obviously improve performance.