ECE 565 homework 2

Js895 **Junqi Sun**

Problem1 :

The final content of cache:
According to the question, we have 8 blocks in total, each of them is 64B. Since its 2-way set associative, we have 4 set, each set consist of 2 blocks.
The final content remains in cache shows below:

Set	**valid**	**cache**
1	**1**	14300 – 1433F
	1	52C00 – 52C3F
2	**1**	CDE40 – CDE7F
	1	F1240 – F127F
3	**1**	92D80 – 92DBF
	0	
4	**1**	ABCC0 – ABCFF
	0	

Result of references:

address	ABCDE	14327	DF148	8F220	CDE4A	1432F	52C22	ABCF2	92DA3	F125C
result	miss	miss	miss	miss	miss	hit	miss	hit	miss	miss

Problem2 :

For L1 cache:	hit : 1 cycle
	Miss: 15 cycles
For L2 cache:	hit: 15 cycles
	Miss: 300 cycles

(a)

AAT = 1 + 3% * (15 + 30% * 300) = 4.15 cycles

(b)

AAT = 1 + 10% * (15 + 5% * 300) = 4 cycles

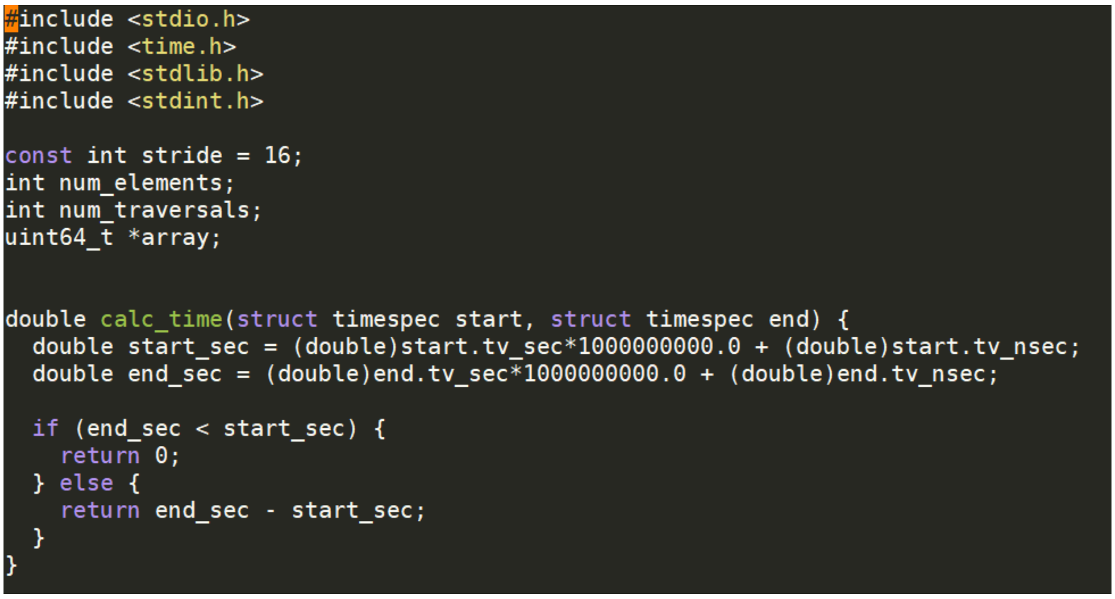
Problem3:

(c)

Instruction of using my test code:

1. **Open the cache_test.c , choose the test part and use /* */ to block other parts that you don't want to test.**
2. **Run ./build.sh**
3. **Run ./cache_test num_elements num_traversals**

Main part source code:



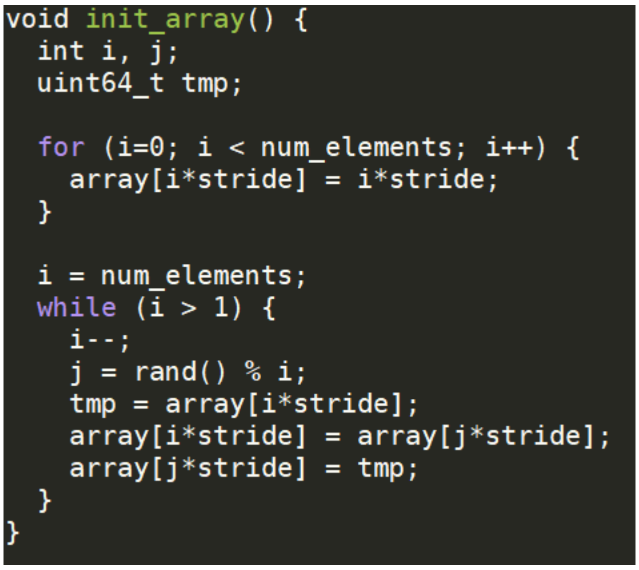
```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <stdint.h>

const int stride = 16;
int num_elements;
int num_traversals;
uint64_t *array;


double calc_time(struct timespec start, struct timespec end) {
  double start_sec = (double)start.tv_sec*1000000000.0 + (double)start.tv_nsec;
  double end_sec = (double)end.tv_sec*1000000000.0 + (double)end.tv_nsec;

  if (end_sec < start_sec) {
    return 0;
  } else {
    return end_sec - start_sec;
  }
}
```

In this part, I still set variables such as stride, num_elements, num_traversals as we did in class.

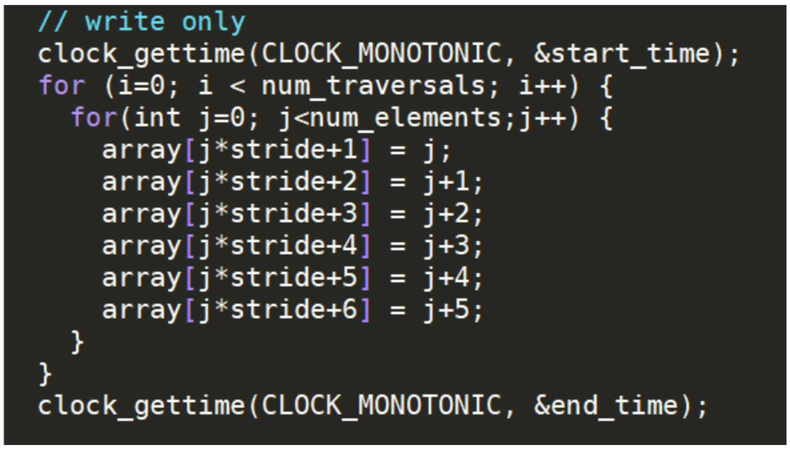


```c
void init_array() {
  int i, j;
  uint64_t tmp;

  for (i=0; i < num_elements; i++) {
    array[i*stride] = i*stride;
  }

  i = num_elements;
  while (i > 1) {
    i--;
    j = rand() % i;
    tmp = array[i*stride];
    array[i*stride] = array[j*stride];
    array[j*stride] = tmp;
  }
}
```

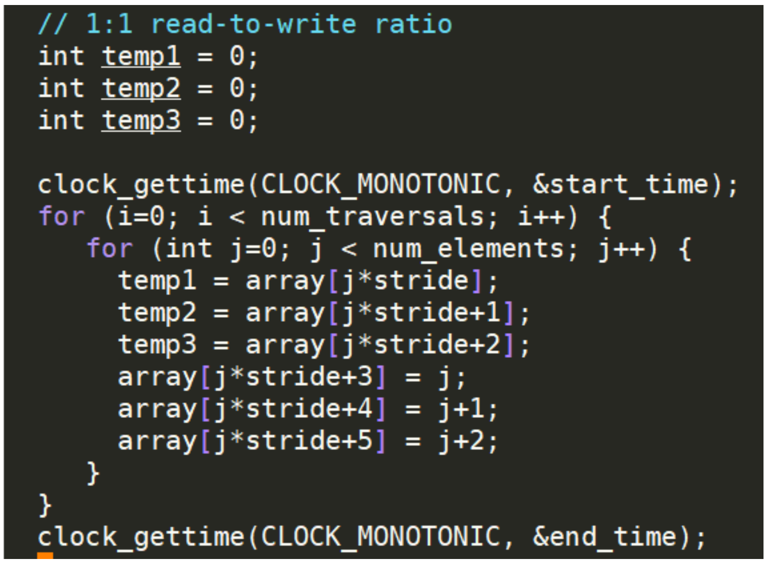
Initialize the array as we did in class.



```c
  // write only
  clock_gettime(CLOCK_MONOTONIC, &start_time);
  for (i=0; i < num_traversals; i++) {
    for(int j=0; j<num_elements;j++) {
      array[j*stride+1] = j;
      array[j*stride+2] = j+1;
      array[j*stride+3] = j+2;
      array[j*stride+4] = j+3;
      array[j*stride+5] = j+4;
      array[j*stride+6] = j+5;
    }
  }
  clock_gettime(CLOCK_MONOTONIC, &end_time);
```

In write only test

I write 6 different variables into 6 different array elements, all instructions are independent. Also we executing the first line, it should be a miss and rest of 5 instructions should be hit on L1 cache. By introducing strides, we can access different blocks in different loop which simulates the real cache applying circumstance.

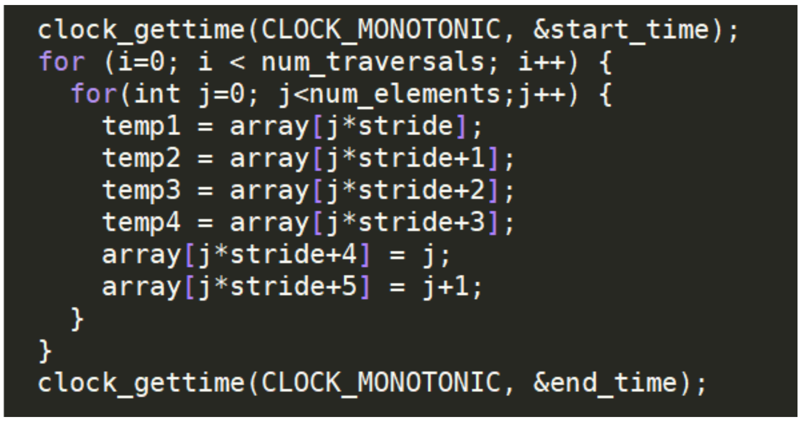


```
// 1:1 read-to-write ratio
int temp1 = 0;
int temp2 = 0;
int temp3 = 0;

clock_gettime(CLOCK_MONOTONIC, &start_time);
for (i=0; i < num_traversals; i++) {
   for (int j=0; j < num_elements; j++) {
     temp1 = array[j*stride];
     temp2 = array[j*stride+1];
     temp3 = array[j*stride+2];
     array[j*stride+3] = j;
     array[j*stride+4] = j+1;
     array[j*stride+5] = j+2;
   }
}
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

In 1:1 read-to-write test

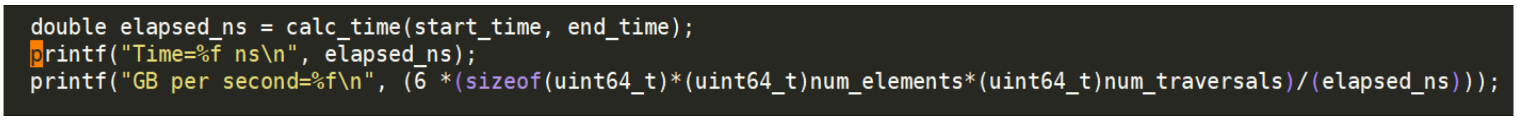
I use 3 variables to act as buffers to read values from 3 different array elements, then load 3 different values to 3 other different elements of array. By using big num_traversals, we could stress the bandwidth that the cache will deal with a lot of independent write and read instructions.



```
clock_gettime(CLOCK_MONOTONIC, &start_time);
for (i=0; i < num_traversals; i++) {
  for(int j=0; j<num_elements;j++) {
    temp1 = array[j*stride];
    temp2 = array[j*stride+1];
    temp3 = array[j*stride+2];
    temp4 = array[j*stride+3];
    array[j*stride+4] = j;
    array[j*stride+5] = j+1;
  }
}
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

In 2:1 read-to-write test

I use 4 variables to act as buffers to read values from 4 different array elements, then load 2 different values to 2 other different elements of array. By using big num_traversals, we could stress the bandwidth that the cache will deal with a lot of independent write and read instructions.



```
double elapsed_ns = calc_time(start_time, end_time);
printf("Time=%f ns\n", elapsed_ns);
printf("GB per second=%f\n", (6 *(sizeof(uint64_t)*(uint64_t)num_elements*(uint64_t)num_traversals)/(elapsed_ns)));
```

To calculate the bandwidth (GB/s, whose numerical value is the same as B/ns), I multiply the total number of instructions we tested in for loop, by the size of each elements. Then divided by the executing time.

My L1 cache is 32K, and each of my elements are 8 bytes.

During the test, I set stride to be 16, num_elements to be 64 (array's size will be 8192B), num_traversals to be 100,000

Testing Type	Write only	1:1 read-write	2:1 read-write
Bandwidth(GB/s)	18.48	46.914	44.175

(d)

Now I set the num_elements to be 320 (array's size will be 40960B, larger than 32K), num_traversals to be 10, 000

Testing Type	Write only	1:1 read-write	2:1 read-write
Bandwidth(GB/s)	16.907	28.247	28.664

According to the result, because the array grows larger than the cache, so it will influence the accessing time for L1 cache, which leads to decreasing of bandwidth. Which is just what I expected.

Problem4:

(b)

Instruction of using my test code:

1. **Run ./build_matrix.sh**
2. **Run ./matrix loop_method**

For loop_method, 0 is for i-j-k, 1 is for j-k-I, 2 is for i-k-j, 3 is for i-j-k tiling

Multiplication methods	I-J-K	J-K-I	I-K-J
Running time(second)	1.802	17.107	0.798

According to lectures, for i-j-k version, we have misses per iteration 1+(element size/block size). For j-k-I version, we have misses per iteration 2. For i-k-j version, we got 2*(element size/block size).

The result show just as I expected, since i-k-j has 1.125 misses per iteration, j-k-I has 2 misses per iteration, i-k-j has only 0.25 misses per iteation. So i-k-j has the shortest running time, i-j-k has the second shortest time of running, while j-k-I has the longest running time.

(d)

Multiplication methods	I-J-K	I-J-K tiling
Running time(second)	1.802	1.602

For my VM, the L2 cache size is 1024KB, 1024 sets and 16 ways_of_associativity, so the block size is 64B. In that case, I choose my tiling block to be 256*256 elements (whose size is 524KB, smaller than 1024KB). The result shows that this loop-tiling optimization did shorten the running time compares to original i-j-k, but not shorter than i-k-j.

I think it's because we fetch for A and B's elements of sub-block's size, some of them could be stored in cache, thus increase the hit rates and decrease the execution time. However, it could not reduce the misses per inner loop instruction as much as i-k-j does, so it will not perform better than i-k-j.