Cache performance and Locality

- By testing the simple example code -

Prepared by

Lee Chang Yoon

32183641@dankook.ac.kr

September 29, 2021

Executive Summary

```
leechangyoon@                          :~/Documents/study/c/system_programming$ ls -l
-rw-rw-r-- 1 leechangyoon leechangyoon   416  9월 28 17:49 cache_hit.c
-rw-rw-r-- 1 leechangyoon leechangyoon   531  9월 29 11:40 cache_miss_1.c
-rw-rw-r-- 1 leechangyoon leechangyoon   530  9월 28 17:49 cache_miss_2.c
-rwxrwxr-x 1 leechangyoon leechangyoon 16768  9월 28 17:37 loop1
-rwxrwxr-x 1 leechangyoon leechangyoon 16776  9월 29 11:41 loop2
-rwxrwxr-x 1 leechangyoon leechangyoon 16768  9월 28 17:38 loop3
```

Figure 1

This report is for reviewing the terms which are used in the cache related basics and the locality which we learned in week 3. In the first section, we will see the meanings of the terms: cache, cache line and cache size. Then, we will use three different codes but do the same job. We can check their names are loop1, loop2 and loop3, according to Figure 1. The first program, loop1, will access the memory near the memory we have referenced before. However, in the second program, loop2, will access the memory little far away from the referenced memory. Also, in the last program, loop3, will access the memory far away from the referenced memory. In this report, we will compare the execution time between those codes and talk about locality, especially spatial locality.

Terms

In this section, we will see the meanings of three terms: cache, cache line, and cache size. First, the cache is the chosen name to represent the level of the memory hierarchy between the processor and main memory. Also, it is used to refer to any storage managed to take advantage of the locality of access. The cache is the most important example of the big idea of prediction. It relies on the principle of the locality to try to find the desired data in the higher levels of the memory hierarchy and provides mechanisms to ensure that when the prediction is wrong it finds and uses the proper data from the lower levels of the memory hierarchy. The hit rates of the cache prediction on modern computers are often higher than 95%. Next, the cache line is the smallest unit of cache operations such as load and flush and continuous word block in the memory. Due to the inefficiency of putting tags to each word, computer some words in a one-word block, which is a cache line. The cache line is valid when it contains cached data or instructions. If not, it is not valid. Last is cache size. Cache size means the size of the cache. If the size of the cache is big, then it increases the recycling rate of the data and gives faster access to the data which is already received. By using several cache mappings, we can efficiently use the cache memory by considering the cache size. In short, we have seen the following terms of the cache-related basics: cache, cache line, and cache size.

Case 1

Loop1

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


int data[0x1000][0x1000][0x10] = {0};

int main() {
        int     sum = 0;
        double  start, end;

        start = (double)clock() / CLOCKS_PER_SEC;
        for (int i = 0; i < 0x1000; i++)
                for (int j = 0; j < 0x1000; j++)
                        for (int k = 0; k < 0x10; k++)
                                sum += data[i][j][k];
        end = (double)clock() / CLOCKS_PER_SEC;

        printf("Program execution time: %lfsec\n", (end -start));
        return 0;
}
```

Figure 2-1

```
leechangyoon@                          :~/Documents/study/c/system_programming$ ./loop1
Program execution time: 0.739596sec
```

Figure 2-2

Let's check loop1. Loop1 is compiled code of cache_hit.c. We can check the code of cache_hit.c in Figure 2-1. What we can find in this code is that loops are accessing the memory in a horizontally sequential way, by just accessing the memory near the referenced memory. For example, let's assume that the address of data[0][0][0] is 0x1000. Then, after the first loop of k, we are now referencing data[0][0][1], which has the address as 0x1004. In this example, we can denote this program has a locality of referencing the nearby memory, which it just referenced before. In short, in this program, the memory of data has a strongest spatial locality among three other programs. Figure 2-2 shows the execution time of loop1.

Case 2

Loop2

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


int data[0x1000][0x1000][0x10] = {0};

int main() {
        int     sum = 0;
        double  start, end;

        start = (double)clock() / CLOCKS_PER_SEC;
        for (int k = 0; k < 0x10; k++)
                for (int i = 0; i < 0x1000; i++)
                        for (int j = 0; j < 0x1000; j++)
                                sum += data[i][j][k];
        end = (double)clock() / CLOCKS_PER_SEC;

        printf("Program execution time: %lfsec\n", (end -start));
        return 0;
}
```

Figure 3-1

```
leechangyoon@                          :~/Documents/study/c/system_programming$ ./loop2
Program execution time: 0.774964sec
```

Figure 3-2

Next, let's check loop2. Loop2 is compiled code from cache_miss_1.c. We can check the code of cache_miss_1.c in Figure 3-1. What we can find in this code is that loops are accessing the memory in a vertically sequential way of j, by accessing the memory little far away from the referenced memory. For example, let's assume that the address of data[0][0][0] is 0x1000. Then, after the first loop of j, we are now referencing data[0][1][0], which has the address as 0x1000 + (0x10) * 4 = 0x1040. In this case, we can denote this program has a locality of referencing the memory far away from the memory we have just referenced before than the program in case1. In short, in this program, the memory of data has a weaker spatial locality than the program in case1. Figure 3-2 shows the execution time of loop2.

Case 3

Loop3

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


int data[0x1000][0x1000][0x10] = {0};

int main() {
        int     sum = 0;
        double  start, end;

        start = (double)clock() / CLOCKS_PER_SEC;
        for (int k = 0; k < 0x10; k++)
                for (int j = 0; j < 0x1000; j++)
                        for (int i = 0; i < 0x1000; i++)
                                sum += data[i][j][k];
        end = (double)clock() / CLOCKS_PER_SEC;

        printf("Program execution time: %lfsec\n", (end -start));
        return 0;
}
```

Figure 4-1

```
leechangyoon@                    :~/Documents/study/c/system_programming$ ./loop3
Program execution time: 3.252602sec
```

Figure 4-2

Next, let's check loop3. Loop3 is compiled code from cache_miss_2.c. We can check the code of cache_miss_2.c in Figure 4-1. What we can find in this code is that loops are accessing the memory in a vertically sequential way of i, by accessing the memory far away from the referenced memory. For example, let's assume that the address of data[0][0][0] is 0x1000. Then, after the first loop of i, we are now referencing data[1][0][0], which has the address as 0x1000 + (0x1000 * 0x1000) * 4 = 0x4001000. In this case, we can denote this program has a locality of referencing the memory far away from the memory we have just referenced before. In short, in this program, the memory of data has a weakest spatial locality among three other programs. Figure 4-2 shows the execution time of loop3.

Conclusion

By checking case1, case 2 and case3, we can find out the difference on efficiency between three different codes of operations which are doing the same job. As we can check from the cases above, in Figure 2-2, Figure 3-2 and Figure 4-2, the program with strong spatial locality is much more efficient than the one which has weak spatial locality.