Reverse Engineering of L1 Cache

Soham Tripathy CS20B073[†], Teja Vardhan CS20B046[‡]

Abstract

To identify the cache block size and the associativity of the L1 cache using reverse engineering.

1. Introduction

In this experiment we will exploit the fact that the cache hit take less latency than the cache miss. Using this we will identify the cache block size and cache associativity.

2. Latency testing

We will be using RDTSC (Read Time-Stamp Counter and Processor) instruction in C inline assembly.

Intel CPUs have a timestamp counter to keep track of every cycle that occurs on the CPU these can be accessed by the RDTSC and RDTSCP assembly instructions.

```
asm volatile ("CPUID\n\t"
 "RDTSC\n\t"
 "mov %%edx, %0\n\t"
 "mov %%eax, %1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
"%rax", "%rbx", "%rcx", "%rdx");
/*********************************/
/*call the function to measure here*/
/*********************************/
asm volatile("RDTSCP\n\t"
 "mov %%edx, %0\n\t"
 "mov %%eax, %1\n\t"
 "CPUID\n\t": "=r" (cycles_high1), "=r" (cycles_low1)::
"%rax", "%rbx", "%rcx", "%rdx");
```

In the code above, the first CPUID call implements a barrier to avoid out-of-order execution of the instructions above and below the RDTS instruction.

The first RDTSC then reads the timestamp register and the value is stored in memory.

The RDTSCP instruction reads the timestamp register for the second time and guarantees that the execution of all the code we wanted to measure is completed.

Finally a CPUID call guarantees that a barrier is implemented again so that it is impossible that any instruction coming afterwards is executed before CPUID itself (and logically also before RDTSCP).

3. CACHE SPECIFICATIONS

First We need to identify the actual specifications of the L1 cache in our system.

For the same purpose we can run the command:

```
    getconf -a | grep CACHE
```

This gave the values as follows:

L1 Cache size = 32KB

L1 Associativity = 8

L1 Block Size = 64B

L1 sets = 64

4. CACHE BLOCK SIZE

We first allocate certain amount of continuous memory (as int arrays) using malloc in C.

We then use the clflush function. The function will flush the block to which the pointer is pointing to, from all levels of the cache hierarchy.

After that we continously access those memory location using a temporary variable and measure the access latency for each request. If the access latencies are more or less equal, the corresponding addresses are from the same cache block. If the access latency for a request to some address Y is very high as compared to that of other addresses, it indicates that the request to address Y is a miss in the cache. Hence, by measuring the difference between two cache miss, we can infer the cache block size.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>

const int till = 128;
const int reptill = 4;

int *p;
int temp;

void clcache()
{
    for(int i = 0; i<till; i++) _mm_clflush(&p[i]);
}

int main() {

    p = (int*) malloc(till*sizeof(int));

    unsigned cycles_high, cycles_low, cycles_high1, cycles_low1;
    uint64_t start, end;

    unsigned int time[till];
    for(int i = 0; i <till; i++) time[i] = 0;

```

```
    clcache();

    for (int j = 0; j < reptill; j++)
    {
        for (int i = 0; i < till; i++)
        {

            asm volatile(
                "CPUID\n\t"
                "RDTSC\n\t"
                "mov %%edx, %0\n\t"
                "mov %%eax, %1\n\t"
                : "=r"(cycles_high), "=r"(cycles_low)::"%rax", "%rbx", "rcx", "%rdx");

            temp = p[i];

            asm volatile(
                "RDTSCP\n\t"
                "mov %%edx, %0\n\t"
                "mov %%eax, %1\n\t"
                "CPUID\n\t"
                : "=r"(cycles_high1), "=r"(cycles_low1)::"%rax", "%rbx", "rcx", "%rdx");

            start = (((uint64_t)cycles_high << 32) | cycles_low);
            end = (((uint64_t)cycles_high1 << 32) | cycles_low1);

            // printf("for index %d: %ld\n", i, end - start);

            time[i] += (end - start);
        }

        clcache();
    }

    for(int i =0; i<till; i++) time[i]/=reptill;

    for(int i = 0; i<till; i++) printf("for the index %d : %u \n", i, time[i]);

    free(p);

    return 0;
}
```

The data of the following experiment is in the "test1.txt" file.

We can clearly observe the cache misses at index 8, 24, 40, and so on. There is a difference of 16. Since, 'int' variable is of size 4B hence, we can take the block size to be 64B.

It matches with the actual specifications.

5. CACHE ASSOCIATIVITY

We have observed from the specifications that the number of sets in the L1 cache is 64. Thus, we will access the blocks belonging to the same set using the above knowledge.

In our approach we will access the first block(0) and check access latency for an element in that block. Now, as we access other blocks from the same set (sets∗blockSize∗i, where i varies from 0 to k), we will keep observing the access latency for the elements in the previously accessed blocks (We will get similar values). If there is spike in the latency, then it implies that we have accessed more blocks in the same set than the associativity and the initially accessed block is replaced. Thus it justifies the high latency when we access an element from the evicted block. This will also give the value of associativity, which is i.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <emmintrin.h>

const int till = 32*64*16;

int *p;

int temp;

void clcache()
{
    for(int i = 0; i<till; i++) _mm_clflush(&p[i]);
}

int main() {

    p = (int*) malloc(till*sizeof(int));

    unsigned cycles_high, cycles_low, cycles_high1, cycles_low1;
    uint64_t start, end;

    for (int i = 0; i < 32; i++)
        {
            temp = p[i*16*64];
        }

    // unsigned int time[32];
    // for(int i = 0; i <32; i++) time[i] = 0;

    clcache();

     for (int i = 0; i < 32; i++)
        {
            temp = p[i*16*64];

            for (int k = 0; k < i+1; k++)
            {
                asm volatile(
```

```
                    "CPUID\n\t"
                    "RDTSC\n\t"
                    "mov %%edx, %0\n\t"
                    "mov %%eax, %1\n\t"
                    : "=r"(cycles_high), "=r"(cycles_low)::"%rax", "%rbx", "rcx", "%rdx");

                temp = p[k*16*64];

                asm volatile(
                    "RDTSCP\n\t"
                    "mov %%edx, %0\n\t"
                    "mov %%eax, %1\n\t"
                    "CPUID\n\t"
                    : "=r"(cycles_high1), "=r"(cycles_low1)::"%rax", "%rbx", "rcx", "%rdx"
    );

                start = (((uint64_t)cycles_high << 32) | cycles_low);
                end = (((uint64_t)cycles_high1 << 32) | cycles_low1);

                printf("for block %d: %ld\n", i, (end - start));

                // time[i] += (end - start);
            }
            printf("\n");
        }

    clcache();

    // for(int i = 0; i<32; i++) printf("for the index %d : %u \n", i, time[i]);

    free(p);

    return 0;
}
```

The data of the following experiment is in the ”test2.txt” file.

We can clearly see the spike at 8. This can infer that the associativity is 8.

This matches with the actual specifications.

References

Intel instructions for latency testing

Sivarama P. Dandamudi. Guide to Assembly Language Programming in Linux, Springer-Verlag, 2005.