

ECE 522

Fall 2024

MP1 Report

**A Timing Experiment with
Memory System**

Lorenzo Bujalil Silva

Machine Specifications

For this assignment, I utilized a Windows PC utilizing WSL2 to be able to utilize the Linux command line. The Linux command line allows me to use the following commands to be able to seamlessly access information on the Memory Hierarchy, Operating System, and CPU. With access to this information, I could confirm my results of my program and perform research on micro-architectural features that could cause variation in memory access time.

CPU:

Command: `lscpu`

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Address sizes:          46 bits physical, 48 bits virtual
Byte Order:             Little Endian
CPU(s):                 20
On-line CPU(s) list:    0-19
Vendor ID:              GenuineIntel
Model name:             13th Gen Intel(R) Core(TM) i7-1370P
CPU family:             6
Model:                  186
Thread(s) per core:     2
Core(s) per socket:     10
Socket(s):              1
```

Memory Hierarchy:

Command: `cat /sys/devices/system/cpu/cpu#/cache/index#/`

Cache Type	Size	Associativity	Cache Line Size	Number of Sets
L1 Instruction Cache	32K	8	64	64
L1 Data Cache	48K	12	64	64
L2 Unified Cache	1280K	10	64	2048
L3 Unified Cache	24576K	12	64	32768

Operating System:

Command: `cat /etc/os-release , cat /proc/version`

```
Operating System: Ubuntu 22.04.3 LTS
Kernel: Linux 5.15.153.1-microsoft-standard-WSL2
```

Compiler:

Command (Removed Optimizations): `gcc -O0 -o msmp1 msmp1.c`

Program Results

Cache Line Size:

Result: 64 Bytes

Function:

```
1  double LineSizeTest(void)
2  {
3      double retval; // Return Value
4      struct timeval t1, t2; // Time Structures
5
6      int stride;
7      int i;
8      int iter;
9
10     int strides[] = {1, 2, 4, 8, 16, 32, 64, 128, 256};
11     double stride_times[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
12                               0.0};
13
14     for(iter = 0; iter < ITER; iter++){
15         for(stride = 0; stride < 9; stride++){
16             populate_caches_with_garbage_data(); // Fill caches with
17             garbage
18             gettimeofday(&t1, NULL);
19             for(i = 0; i < MAX_N; i+=strides[stride]){
20                 array[i]++;
21             }
22             gettimeofday(&t2, NULL);
23             stride_times[stride] += elapsedTime(t1,t2);
24         }
25     }
26
27     for(stride = 0; stride < 9; stride++){
28         stride_times[stride] /= ITER;
29         printf("Stride: %d, Access Time: %lf ms\n", strides[stride],
30               stride_times[stride]);
31     }
32
33     for(stride = 0; stride < 8; stride++){
34         if(stride_times[stride] - stride_times[stride+1] < 1){
35             retval = (double)strides[stride+1];
36         }
37     }
38
39     return retval;
40 }
```

Program Output:

```
Starting Cache Line Test...
Actual LLC Cache Line Size: 64.000000
Stride: 1, Access Time: 91.380200 ms
Stride: 2, Access Time: 43.841200 ms
Stride: 4, Access Time: 22.381000 ms
Stride: 8, Access Time: 12.102300 ms
Stride: 16, Access Time: 7.028400 ms
Stride: 32, Access Time: 5.665400 ms
Stride: 64, Access Time: 4.958700 ms
Stride: 128, Access Time: 3.848500 ms
Stride: 256, Access Time: 2.285600 ms
Last Level Cache Line Size: 64.000000
```

Last Level Cache Size:

Result: 24576 KB \approx Greater than 16384 KB and Less than 32768 KB

Function:

```
1 double CacheSizeTest(void)
2 {
3     double retval;
4     struct timeval t1, t2;
5     int i;
6     int iter;
7     int cache;
8     int num_caches = 13;
9     int cache_sizes[] = {16 * KB, 32 * KB, 64 * KB, 128 * KB, 256 *
10                          KB, 512 * KB,
11                          1024 * KB, 2048 * KB, 4096 * KB, 8192 * KB,
12                          16384 * KB,
13                          32768 * KB, 65536 * KB};
14     double cache_times[] = {0.0, 0.0, 0.0, 0.0, 0.0,
15                             0.0, 0.0, 0.0, 0.0, 0.0,
16                             0.0, 0.0, 0.0};
17
18     int cache_line_size = LLC_CACHE_LINE_SIZE; // We know that the
19     cacheline size is 64 Bytes from previous tests
20
21     for(iter = 0; iter < ITER; iter++){
22         for(cache = 0; cache < num_caches; cache++){
23
24             // Load Caches with Garbage
25             populate_caches_with_garbage_data();
26
27             // Load potential cache size by reading in one cache line
28             in at a time
29             for(i = 0; i < cache_sizes[cache]; i+=64){
30                 array[i]++;
31             }
32
33             gettimeofday(&t1, NULL);
34             // Measure time to access data
35             for(i = 0; i < cache_sizes[cache]; i+=64){
```

```

32         array[i]--;
33     }
34     gettimeofday(&t2, NULL);
35
36     cache_times[cache]+=elapsedTime(t1,t2);
37 }
38 }
39
40 for(cache = 0; cache < num_caches; cache++){
41     cache_times[cache] /= ITER;
42     printf("Cache Size: %d KB, Access Time: %lf\n", cache_sizes[
43         cache] / KB, cache_times[cache]);
44 }
45
46 double prev_slowdown = 0;
47 double curr_slowdown;
48 for(cache = 1; cache < num_caches; cache++){
49     curr_slowdown = cache_times[cache] / cache_times[cache-1];
50     #if LOG_SLOWDOWN
51     printf("Slowdown from %d KB to %d KB: %lf\nPrev Slowdown: %lf\
52         nCurr Slowdown: %lf\n", cache_sizes[cache-1] / KB,
53         cache_sizes[cache] / KB, cache_times[cache]/cache_times[
54         cache-1],
55         prev_slowdown, curr_slowdown);
56     #endif
57     if(curr_slowdown > prev_slowdown){
58         retval = cache_sizes[cache-1] / KB;
59     }
60     prev_slowdown = curr_slowdown;
61 }
62
63 return retval;
64 }

```

Program Output:

```

Location of Test Array: 0x56076a1f9040
Location of Garbage Array: 0x56076e1f9040
Starting Cache Size Test...
Actual L1 Cache Size: 48.000000 KB
Actual L2 Cache Size: 1280.000000 KB
Actual LLC Cache Size: 24576.000000 KB
Cache Size: 16 KB, Access Time: 0.000500
Cache Size: 32 KB, Access Time: 0.000900
Cache Size: 64 KB, Access Time: 0.001200
Cache Size: 128 KB, Access Time: 0.002600
Cache Size: 256 KB, Access Time: 0.006500
Cache Size: 512 KB, Access Time: 0.010400
Cache Size: 1024 KB, Access Time: 0.023500
Cache Size: 2048 KB, Access Time: 0.079600
Cache Size: 4096 KB, Access Time: 0.168100
Cache Size: 8192 KB, Access Time: 0.420800
Cache Size: 16384 KB, Access Time: 0.932700
Cache Size: 32768 KB, Access Time: 2.430000
Cache Size: 65536 KB, Access Time: 5.642300
Last Level Cache Size: Greater than 16384.000000 KB and less than 32768.000000 KB

```

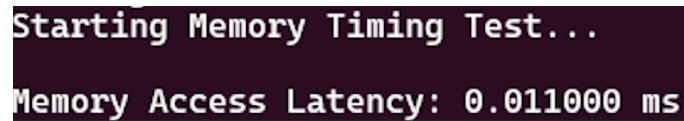
Memory Access Latency:

Result: 0.01 ms

Function:

```
1 double MemoryTimingTest(void)
2 {
3     double retval;
4     struct timeval t1, t2;
5     int i;
6
7     // Populate the last level cache
8     for(i = 0; i < LLC_CACHE_SIZE; i++){
9         array[i]++;
10    }
11
12    // Access a point in memory not within the last level cache
13    // This will cause a LLC Miss and will need to go to memory to
14    // retrieve the data
15    // This then means that the time to access this element will be
16    // the memory latency
17    gettimeofday(&t1, NULL);
18    array[MAX_N-1]++;
19    gettimeofday(&t2, NULL);
20
21    retval = elapsedTime(t1,t2);
22
23    return retval;
24 }
```

Program Output:

A terminal window with a dark background and light-colored text. The first line reads "Starting Memory Timing Test..." and the second line reads "Memory Access Latency: 0.011000 ms".

```
Starting Memory Timing Test...
Memory Access Latency: 0.011000 ms
```

Last Level Cache Associativity:

Result: 12

Function:

```
1
2 double CacheAssocTest(void)
3 {
4     double retval;
5     struct timeval t1, t2;
6
7     // L1_NUMBER_OF_SET_BITS = log2(64) = 6
8     // L1_NUMBER_OF_OFFSET_BITS = log(64) = 6
9     // (52 Tag Bits) | (6 Set Bits) | (6 Offset Bits)
10
11     // L2_NUMBER_OF_SET_BITS = log2(2048) = 11
12     // L2_NUMBER_OF_OFFSET_BITS = log(64) = 6
13     // (47 Tag Bits) | (11 Set Bits) | (6 Offset Bits)
14
15     // LLC_NUMBER_OF_SET_BITS = log2(32768) = 15
16     // LLC_NUMBER_OF_OFFSET_BITS = log2(64) = 6
17     // (43 Tag Bits) | (15 Set Bits) | (6 Offset Bits)
18
19     double associativities[37];
20
21     int assoc_i;
22     for(assoc_i = 0; assoc_i < 37; assoc_i++){
23         associativities[assoc_i] = 0.0;
24     }
25
26     int i;
27     int assoc;
28     int iter;
29
30     for(iter = 0; iter < ITER; iter++){
31         for(assoc = 0; assoc <= 36; assoc++){
32             populate_caches_with_garbage_data();
33             // Populate set 0 of the L1 cache
34             for(i = 0; i <= assoc; i++){
35                 array[2*i << 12]++;
36             }
37             // If the assoc we are testing is less than the actual
38             // associativity
39             // Then we will see fast access times. On the other hand,
40             // a higher associativity
41             // would result in slower access times.
42             gettimeofday(&t1, NULL);
43             for(i = 0; i <= assoc; i++){
44                 array[2*i << 12]--;
45             }
46             gettimeofday(&t2, NULL);
47
48             associativities[assoc] += elapsedTime(t1,t2);
49         }
50     }
51
52     double max_assoc_time = 0;
```

```

52     for(assoc = 0; assoc < 37; assoc++){
53         associativities[assoc] /= ITER;
54         printf("Associativity: %d, Access Time: %lf ms\n", assoc,
55             associativities[assoc]);
56         if(associativities[assoc] > max_assoc_time) {
57             max_assoc_time = associativities[assoc];
58             retval = assoc;
59         }
60     }
61     return retval;
62 }

```

Program Output:

```

Starting Cache Associativity Test...
Associativity: 0, Access Time: 0.000000 ms
Associativity: 1, Access Time: 0.000000 ms
Associativity: 2, Access Time: 0.000000 ms
Associativity: 3, Access Time: 0.000200 ms
Associativity: 4, Access Time: 0.000100 ms
Associativity: 5, Access Time: 0.000200 ms
Associativity: 6, Access Time: 0.000100 ms
Associativity: 7, Access Time: 0.000100 ms
Associativity: 8, Access Time: 0.000100 ms
Associativity: 9, Access Time: 0.000200 ms
Associativity: 10, Access Time: 0.000000 ms
Associativity: 11, Access Time: 0.000100 ms
Associativity: 12, Access Time: 0.000400 ms
Associativity: 13, Access Time: 0.000000 ms
Associativity: 14, Access Time: 0.000100 ms
Associativity: 15, Access Time: 0.000000 ms
Associativity: 16, Access Time: 0.000000 ms
Associativity: 17, Access Time: 0.000300 ms
Associativity: 18, Access Time: 0.000100 ms
Associativity: 19, Access Time: 0.000300 ms
Associativity: 20, Access Time: 0.000100 ms
Associativity: 21, Access Time: 0.000100 ms
Associativity: 22, Access Time: 0.000200 ms
Associativity: 23, Access Time: 0.000200 ms
Associativity: 24, Access Time: 0.000000 ms
Associativity: 25, Access Time: 0.000100 ms
Associativity: 26, Access Time: 0.000200 ms
Associativity: 27, Access Time: 0.000200 ms
Associativity: 28, Access Time: 0.000100 ms
Associativity: 29, Access Time: 0.000200 ms
Associativity: 30, Access Time: 0.000200 ms
Associativity: 31, Access Time: 0.000400 ms
Associativity: 32, Access Time: 0.000100 ms
Associativity: 33, Access Time: 0.000200 ms
Associativity: 34, Access Time: 0.000100 ms
Associativity: 35, Access Time: 0.000100 ms
Associativity: 36, Access Time: 0.000100 ms
Last Level Cache Associativity: 12.000000

```


Reasoning of Results:

Populate Caches with Garbage Data:

I felt that it was important to note that at the beginning of the tests, I ensured to load the caches with useless data with an array of size bigger than the last level cache to ensure that the initial access to memory for the timing would be a miss. Moreover, when attempting to guess the cache parameters, it is important to repopulate this data each time, to ensure that previous loads into the cache won't affect future timing.

Cache Line Size:

Linear Memory Accesses

When going about calculating the cache line size, an ideal approach is to create an array large enough to be the size of a L1 cache, then iterate through each byte of this array and time the access each of these bytes. Initially, you will get a compulsory miss that pulls the entire cacheline into the L1 cache. Ideally, the access times after this initial miss for the first cacheline should be the same. At a certain point you will cross into the next cacheline and here the access time will be a bit slower as you will miss and need to go deeper into the memory hierarchy to retrieve the next cacheline. At this point you would know which access caused the higher latency, and this point would be the size of the cacheline.

However, there are various issues with this approach. Modern processors have prefetchers that retrieve multiple lines at once. Linearly probing memory would result in a constant hit in the L1 cache, preventing any delay in access time.

Strided Memory Accesses

One way to circumvent this is to implement a strided memory access pattern. We could set up a loop that tries various potential strides, and strides less than the size of the cacheline have few misses and various hits in the same cacheline. As you increase this stride, you tend to have more and more misses, until the stride you are testing is the size of the cacheline and you are missing 100% of the time. As you go on further strides, they will continue to miss constantly as they are constantly accessing different cachelines.

According to the structure of the loop, as you increase the strides, you should be accessing less elements in the array, which means that the time to stride through the array should go down as stride increases. When you get to the size of the cacheline, you should see a time delay spike. This will signal the size of the cacheline.

An issue that arises from using this approach, is the strided prefetcher implementation that is able to detect strides in memory accesses, then prefetch those cachelines ahead of time. This would cause some diminishing effects on the

access times for our timing, however, after running my implementation, there was a clear increase on the correct cacheline size.

Other Considerations

Some other considerations, that I didn't have to time to integrate would be potential out of order execution issues. Where potentially stores and loads could be made into memory at the same time which would reduce the overall delay to access memory.

Last Level Cache Size:

In order to find the cache size, we first would iterate through a list of potential cache sizes and access elements within a larger array for the subsection of this cache size. Then, if the size of the memory we accessed was smaller than the cache. We would be able to re-access those same memory addresses and we would have lower latency to constantly have hits in the cache. At a certain point in our probing of the possible cache sizes, we will see a bit of a larger delay. This would mean that we have read more than the cache size, and there is a need to go to the next level to retrieve the data. Effectively, you would see some delay when going from L1 to L2, then L2 to L3, then finally L3 to main memory.

In my results, I was able to see a clear bump at the L1 to L2, L2 to L3, and L3 to main memory. This was able to give me a clear sign of the total size of the last level cache. I effectively iterated through the list of access times and generated a ratio of slow down between accesses, and would determine a threshold to update the size. There was one final issue, which was that I could have possibly skipped the size of the cache, which means that it must be somewhere between the current and previous accesses. This would give me a range consistent with the actual size of the LLC.

Memory Access Latency:

In order to perform the memory access latency test, I first accessed an entire array that was the size of the last level cache. This would completely populate the last level cache. I wanted it to be completely full, so that when I would re-access a memory location that was not in the address space of this array, I would have a miss on all levels of the memory hierarchy and would need to access main memory. I then recorded the time of this final access, and that would give me the latency for memory access.

I also thought of a potential other way to do this, which is to measure the time to access a compulsory miss. This effectively will miss on all levels as well and need to access main memory.

Last Level Cache Associativity:

Finally, in order to calculate the LLC associativity, I needed to use information on the number of sets, and the cacheline size. This information would give me the set bits and the offset bits to be able to index into the caches. With knowing the information about these bits, I would be able to make memory accesses to locations in memory where they all have the same set. I would be able to implement this by ensuring that all accesses are made to set 0 and then I would change the tag bits to ensure that a different way would be used.

Eventually, all the ways would be used and a line would need to be evicted. This would cause a higher delay to access lower levels of memory and depending on the protocols for the hierarchy will cause a spike in latency. I can do the same as before with the cache size, and access these various locations in memory with the same set once to bring them in, and then if I brought in less than the associativity I would be able to quickly read from them. At the point I reach the associativity, I would have a higher access time, which is what I used to determine the correct associativity of the last level cache.