CSC367 assignment 1

hongmao.wu(1008134207)

October 2023

1 Know your machine's memory system

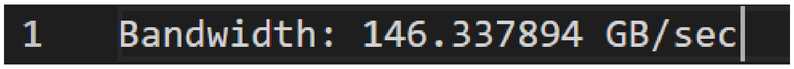
1.1 Design an experiment to measure the memory bandwidth in your computer

Memory bandwidth is defined as the largest amount of data transferred over the amount of time taken to transfer the data. In this experiment, the memory bandwidth unit is gigabytes per second.

This paragraph explains how computers write to main memory. A cache comprises multiple cache lines, with each cache line storing 64 bytes of data. In computer architecture, caches are designed so that when a cache becomes full, one cache line is evicted to make room for new data. Prior to eviction, the computer writes the data stored in the cache line back to the main memory.

This paragraph outlines the rationale behind the experiment. When a program attempts to issue write operations that target contiguous memory addresses, the computer initially stores the data in its cache before updating the data in the main memory. Consequently, executing multiple write operations that address memory locations byte by byte may not efficiently demonstrate the maximum write bandwidth. The main memory may not receive write requests as rapidly as desired. Therefore, to accurately measure write bandwidth, the experiment necessitates evicting one cache line per memory write operation. Assuming each cache line comprises 64 bytes, each memory write operation must target a memory address that is 64 bytes apart from the address accessed in the previous memory write operation.

This paragraph outlines the process of the experiment. First, the program initializes an array to represent a sufficient amount of memory space. Second, it fills the L1, L2, and L3 caches by looping through the array and assigning values. Third, the program issues a multitude of memory write operations within a for loop, performing a write operation in each iteration. Finally, it calculates the total amount of data transferred and divides this result by the total amount of time taken.



```
Bandwidth: 146.337894 GB/sec
```

Figure 1: Bandwidth

Figure 1 displays a typical result of this experiment. The maximum bandwidth is approximately 146 GB/s. Notice the number might change a little bit depending on the status of computer. The result is recorded in the file `Bandwidth.txt`.
This paragraph explains the results and associated factors. Firstly, it's important to note that the for loop itself requires some time to calculate indexes and compare values, and this time is incorporated into the total time measurement. However, these operations typically consume a relatively shorter amount of time compared to memory operations. Secondly, the final result can be influenced

by two key factors: the quantity of write operations performed and the size of the array. If the experiment does not repeat the same memory operations a sufficient number of times, the result may exhibit instability.

1.2 Design an experiment to determine the number of levels in the CPU cache hierarchy and their sizes and latency

This paragraph outlines the characteristics of L1, L2, and L3 caches. Cache latency refers to the time it takes for a computer to retrieve data from a cache. L1 cache, which is the closest cache to the central processing unit (CPU) in a computer, boasts the smallest storage capacity and the lowest cache latency. In contrast, L2 cache offers more storage capacity than L1 and L3 cache provides the largest storage space but has the highest cache latency.

This paragraph explains the core concepts behind the experiment. When the CPU cannot locate data in a lower-level cache, such as the L1 cache, it proceeds to fetch the data from a higher-level cache or the main memory. However, this process introduces additional cache latency. The experiment leverages this property to reveal both cache size and cache latency. If the cache has a capacity of X for storing data, any memory access operation accessing data within that X capacity would result in a cache hit. Based on this idea, the experiment incrementally increases X until the total cache latency begins to spike significantly. This point signals that the CPU needs to fetch data from a higher-level cache, resulting in increased cache latency per memory operation. In essence, when X surpasses a certain cache size threshold, every memory operation necessitates the CPU to retrieve a cache line from a higher-level cache. Therefore, X effectively denotes the cache size.

This paragraph outlines the process of the experiment. First, the program creates three lists; one list for one cache. Each list contains a sequence of numbers representing estimated cache sizes for the respective cache level. Second, the program conducts approximately 10,000 tests for each estimated cache size, with each test involving 10,000 memory operations. Third, the program calculates the total elapsed time by summing the time taken by each test.

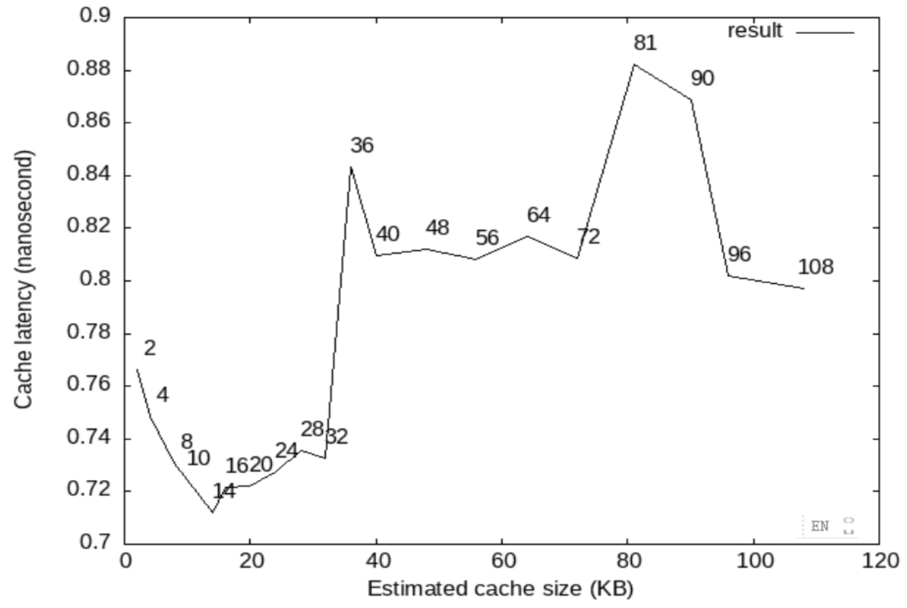


Figure 2: L1 size

Cache sizes are typically designed to be a power of 2. In Figure 2, the L1 cache size is 32 KB, and its latency is approximately 0.74 nanoseconds. Beyond 36 KB, cache latency undergoes a substantial increase, signifying a surge in L1 cache misses. The system experiences additional time overhead as it retrieves cache lines from L2. This result is in ”L1 cache size.png”.

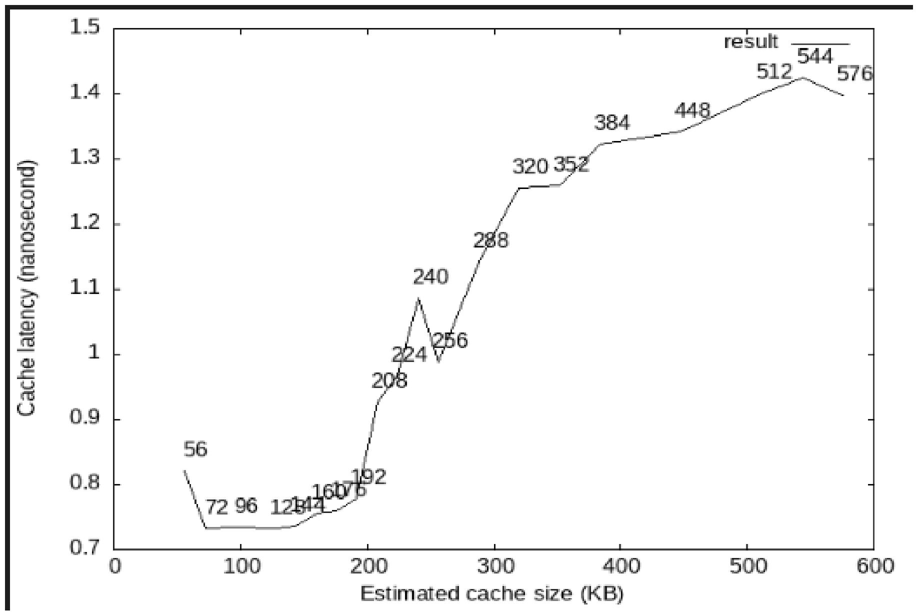


Figure 3: L2 size

Figure 3 provides a less distinct result, but it suggests that the L2 cache size is 256 KB, with an approximate latency of 0.9 nanoseconds. This conclusion is reached because if L2 cache size were 128kb, the experiment would show a noticeable latency increase after 128 KB, which is not observed. Similar reasoning could be applied to 512kb to know L2 cache cannot be 512kb. This result is in "L2 cache size.png".

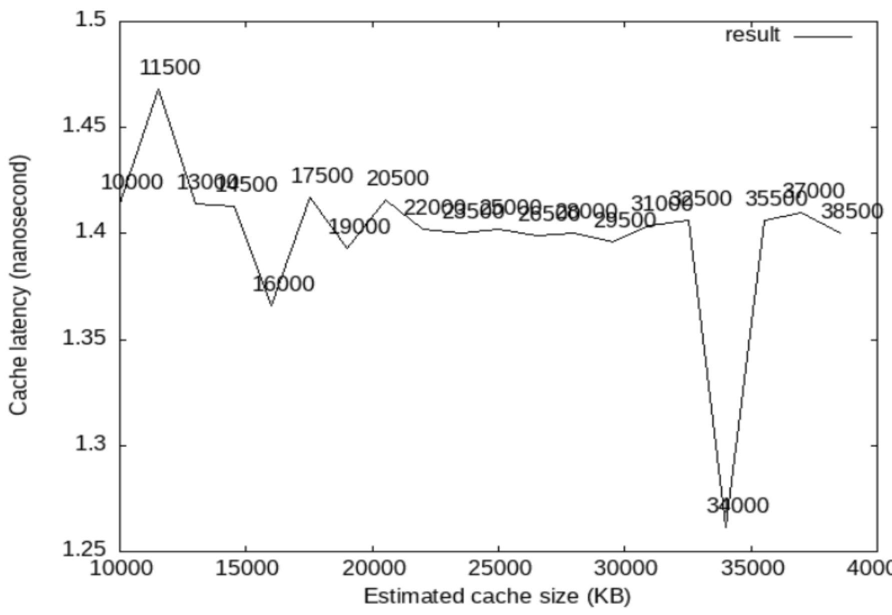
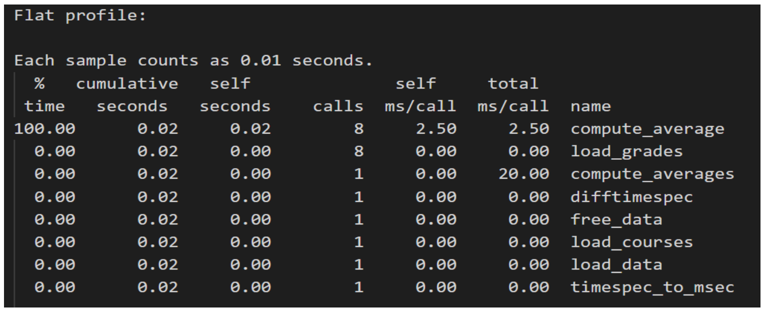


Figure 4: L3 size

Figure 4 lacks a noticeable pattern despite a wide range of estimates. However, it's important to note that this conclusion may be wrong. The experiment could be influenced by other factors, including cache eviction policies, memory access patterns, page faults, and more. This result is in "L3 cache size.png".

2 Performance and Profiling



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
100.00      0.02     0.02        8     2.50     2.50  compute_average
  0.00      0.02     0.00        8     0.00     0.00  load_grades
  0.00      0.02     0.00        1     0.00    20.00  compute_averages
  0.00      0.02     0.00        1     0.00     0.00  difftimespec
  0.00      0.02     0.00        1     0.00     0.00  free_data
  0.00      0.02     0.00        1     0.00     0.00  load_courses
  0.00      0.02     0.00        1     0.00     0.00  load_data
  0.00      0.02     0.00        1     0.00     0.00  timespec_to_msec
```

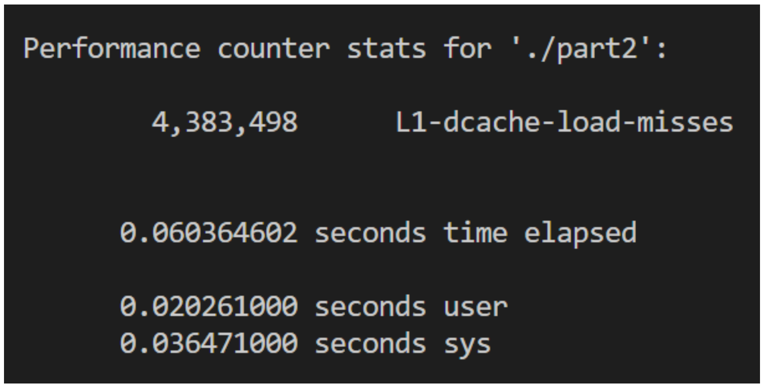
Figure 5: gprof result for ./part2

This paragraph discusses the profiling results and code analysis. Profiling ./part2 with gprof generates the result displayed in Figure 5, which indicates that the program spends the majority of its runtime in the `compute_average` function. This observation is supported by the fact that `compute_average` is called `courses_count` times within a for loop in the code.

```
for (int i = 0; i < courses_count; i++) {
    compute_average(&(courses[i]));
}
```

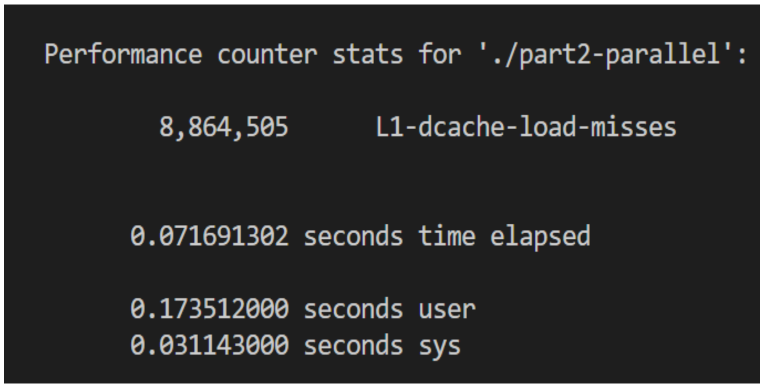
This session of code is parallelizable. Each iteration could be replaced by a thread, which means there will be eight threads running concurrently.

This paragraph explains key results observed from the perf report. Perf is used to record cache misses in ./part2 and ./part2-parallel. These results are shown in Figures 6 and 7. The number of L1 data load misses in ./part2-parallel is much greater than the L1 data load misses of ./part2. Also, the time taken by ./part2-parallel is slightly higher than ./part2.



```
Performance counter stats for './part2':

       4,383,498      L1-dcache-load-misses


     0.060364602 seconds time elapsed

     0.020261000 seconds user
     0.036471000 seconds sys
```

Figure 6: cache misses count without parallelization



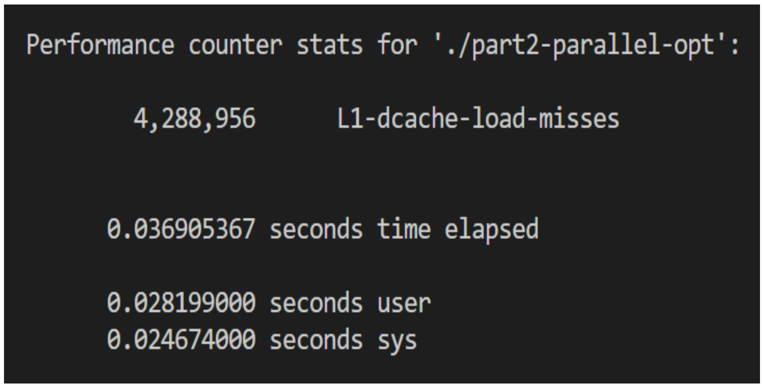
```
 Performance counter stats for './part2-parallel':

         8,864,505      L1-dcache-load-misses


       0.071691302 seconds time elapsed

       0.173512000 seconds user
       0.031143000 seconds sys
```

Figure 7: cache misses count with parallelization

The cause of the high number of cache misses is related to false sharing. Inside ./part2-parallel, a session of code inside `compute_average` method listed below causes false sharing.

```
the_course->average = 0.0;
for (int i = 0; i < the_course->grades_count; i++) {
    the_course->average += the_course->grades[i].grade;
}
the_course->average /= the_course->grades_count;
```

This paragraph explains how the code causes false sharing and suggests optimization. Each thread needs to update `The_course->average` a total of `grades_count` times. `The_course->average` of each course is stored close to each other in memory. If one thread updates its `The_course->average`, the system will notify other threads to invalidate cached data, resulting in a high number of cache misses.
After using a local variable to store the intermediate value and assign the final value to `The_course->average`, the number of cache misses is significantly reduced. The result can be seen in Figure 8.



```
Performance counter stats for './part2-parallel-opt':

        4,288,956      L1-dcache-load-misses


      0.036905367 seconds time elapsed

      0.028199000 seconds user
      0.024674000 seconds sys
```

Figure 8: cache misses count with parallelization and optimization