CSC367: Assignment 1

Faisal Shaik

August 2024

1 Memory Bandwidth Experiment

1.1 Introduction

This experiment pushes the memory bus to it's limit by reading and writing to a large array. The goal is to measure the memory bandwidth of the system. The experiment was conducted on the DH 2020 lab machines. The code for the experiment can be found in the `part1/part1a.c` file.

1.2 Methods

This code measures a computer's memory write bandwidth by performing a large number of memory writes and calculating the write speed. It allocates a huge matrix using `posix_memalign`, ensuring alignment with cache lines for efficient access. The `write_to_memory` function fills this matrix using either standard `memset` or SIMD instructions for potentially faster writes.

The main function, `measure_memory_write_bandwidth`, times how long it takes to write to the entire matrix. It allocates 100 million rows, writes to them, and then calculates the bandwidth by dividing the total data written by the time taken. Each row is 5 cache line sizes long, so $5 \times 64 = 320$ bytes. We write an entire row at each iteration using `memset`. The large size of each row will overwhelm cache, forcing writes to main memory and thus measuring true memory bandwidth rather than cache speed.

This test is designed to bypass all levels of cache (L1, L2, L3) by using row sizes larger than cache lines and a very high number of rows. The result provides a practical measure of the system's maximum memory write speed, which is useful for understanding performance limitations and optimizing memory-intensive applications.

1.3 Results & Discussion

Interestingly, this experiment found the memory write bandwidth to be **3.2 GB/s**. This is quite low, in fact we expected it to be **32 GB/s** for the DH 2020 lab machines. Initially, it was suspected that there was some overhead in the `memset` function which was causing the low bandwidth. Using `_mm_store_si128` produced identical results, so this was ruled out. Repeated experiments at varying times suggests that the low bandwidth is not due outlier measurements. At this point in time, it remains unclear why the bandwidth is so low.

2 Cache Hierarchy Experiment

2.1 Introduction

Computers have a hierarchy of memory systems, with the fastest and smallest memory being the CPU registers and the slowest and largest memory being the hard drive. The cache is a small, fast memory that is used to store frequently accessed data. This report details an experiment that measures the performance of a cache hierarchy using a simple matrix. We seek to determine the different cache sizes and latencies for the cache levels in the hierarchy.

2.2 Methods

The experiment was conducted on the DH 2020 lab machines. These featured a **64 byte cache line**. The method used in this experiment can be broken down into two steps,

a. **Fill Up Cache Levels:** We loop through a large matrix that will, by assumption, be large enough to fill up the cache levels and further invade direct memory. At each iteration i, we have a "working set" of data; the first i rows of the matrix. In each iteration, we loop through every row in our working set and access the middle element, to "load" the rows into the cache. This gradually fills up the cache levels as the working set grows.

b. **Measure Access Time:** After touching i rows in the working set at the i^{th} iteration, we then write 64 bytes of data to the first row of the matrix. We measure how long this write takes, which is a measure of the write latency.

The code for the experiment as described above can be found in the `part1/part1b.c` file. The idea is that at each iteration, the working set will fill more and more of the cache levels, and eventually the first row of the matrix will be evicted from the cache levels, causing a cache miss and yielding a higher latency. The matrix had **262144 rows**, each row being 64 bytes long, for a total size of **16 MiB**.

Data Collection:

We can then plot the latency against the size of the working set to determine the cache sizes and latencies for the cache levels in the hierarchy as shown in the results section. The code for generating the graphs can be found in the `part1/gen/generate_graphs.py` file. The plots were generated in the following steps:

a. The data was collected by running the `part1/part1b.c` program using `make run`. The `make run` command invokes a bash script `myscript.sh` that runs the C executable which generates a CSV file containing the latency and working set size data.

b. The CSV file was then read and plotted using the `part1/gen/generate_graphs.py` script, which is invoked by `myscript.sh`. The script first makes sure that all the necessary Python packages are installed. Specifically, `matplotlib` and `pandas` are used to generate the scatter plots and box plots.

2.3 Results

Below are the results of the experiment. All graphs were generated using the `part1/gen/generate_graphs.py` script.

For the scatter plots in *Figure 1* and *Figure 2*, the x-axis represents the size of the working set in bytes, and the y-axis represents the latency in nanoseconds. Both plots shows the latency of writing to the first row of the matrix as the working set grows.

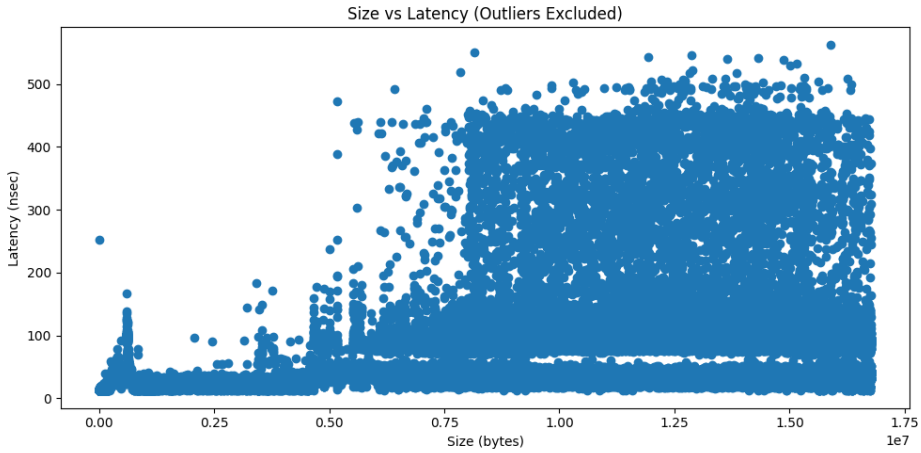

Figure 1: Latency vs. Working Set Size

Here is another plot better showing the density, and also provides an average line for the data.

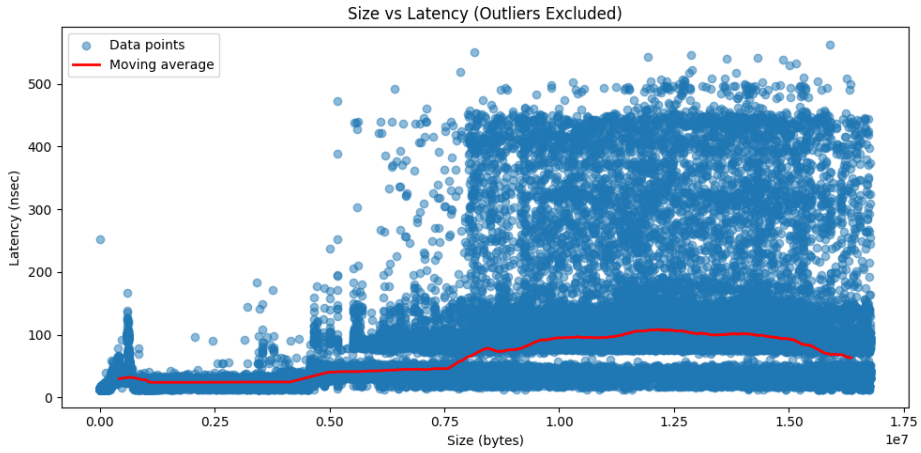

Figure 2: Latency vs. Working Set Size with Average Line

Here is a box plot of the data, showing the distribution of the latencies at three different intervals corresponding to the patterns in the scatter plots.

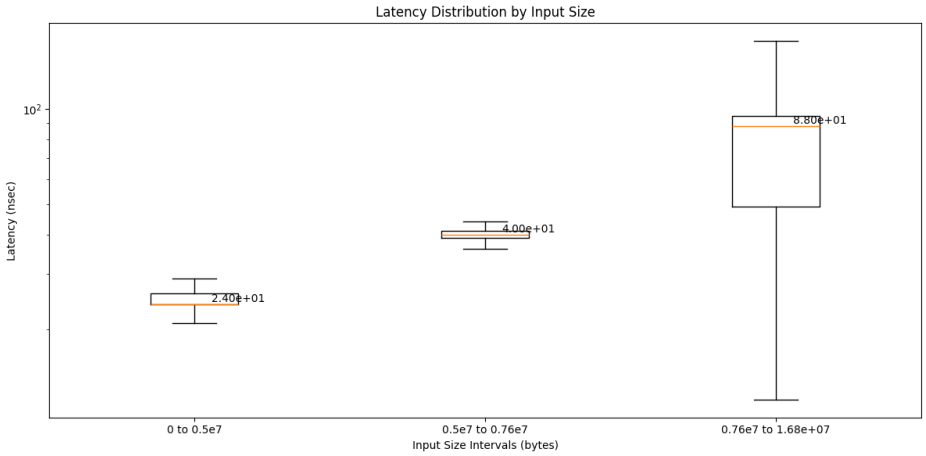

Figure 3: Box Plot of Latencies

2.4 Discussion

The plot in *Figure 1* shows a clear pattern of increasing latency as the working set grows. However, it is clear that there is a lot of noise in the data. We have many low latencies at even large working sets.

That being said, we can still observe a "staircase-like" behaviour, where the latencies seem to jump to a new upper bound after passed certain size thresholds. The box plot in *Figure 3* has been made corresponding to these thresholds. From 0 to 5×10^6 bytes, the average latency is 24 nanoseconds, from 5×10^6 to 7.6×10^6, the average latency is 40 nanoseconds, and from 7.6×10^6 to 1.69×10^7, the average latency is 88 nanoseconds.

However, if look at the upper bounds instead while ignoring outliers then we have the following:

- From 0 to 5×10^6 bytes, the upper bound is 30 nanoseconds.
- From 5×10^6 to 7.6×10^6, the upper bound is 130 nanoseconds.
- From 7.6×10^6 to 1.69×10^7, the upper bound is 460 nanoseconds.

This could possibly correlate to the L1, L2, and L3 cache levels respectively. However, we never got to see another bump, which we would expect for the main memory. This could be due to the fact that the matrix was not large enough to fill up till main memory.

A possible explanation for the large amount of noise in the data could be due to inconsistent cache eviction. Perhaps the cache eviction policy is not as simple as we assumed, and does not always evict like a FIFO queue. This could explain why we see low latencies at large working sets, as the cache may not always evict the first row of the matrix.

3 Performance & Profiling

3.1 Introduction

In this part of the assignment, we are tasked with parallelizing a given code that calculates the course average for many classes. After parallelizing the code, we further optimize it after running `perf`. Throughout this entire process, we compare the performance of the parallel and parallel-optimized code with the original serial code.

3.2 Methods

Note that we limited the number of spawned threads to be only 8 in the parallelized code. This was done because the DH 2020 lab machines only have 8 cores, and we wanted to avoid oversubscription. The code for the parallelized and optimized code can be found in the `part2/part2-parallel.c` file.

The first attempt of parallelizing the code could be broken down into two main ideas:

a. **When Courses $>$ Threads:** Then each thread may do one or more courses.

b. **When Courses $\leq$ Threads:** Then multiple threads can work on the same course.

The code was then optimized by aligning the data structures to cache lines, specifically the `thread_args struct`. Moreover, the second case of the parallel code was discarded as the overhead of synchronization was too high, at least for the given data generation. More on this in results.

3.3 Results & Discussion

Running `gprof` on the serial code revealed the following information:

```
Flat profile:

    Each sample counts as 0.01 seconds.
    %   cumulative   self              self     total
    time   seconds   seconds    calls  ms/call  ms/call  name
    100.00      0.04     0.04        8     5.00     5.00  compute_average
    0.00      0.04     0.00        8     0.00     0.00  load_grades
    0.00      0.04     0.00        1     0.00    40.00  compute_averages
    0.00      0.04     0.00        1     0.00     0.00  difftimespec
    0.00      0.04     0.00        1     0.00     0.00  free_data
    0.00      0.04     0.00        1     0.00     0.00  load_courses
    0.00      0.04     0.00        1     0.00     0.00  load_data
    0.00      0.04     0.00        1     0.00     0.00  timespec_to_msec

    %         the percentage of the total running time of the
    time       program used by this function.
```

As we ccan see, the `compute_average` function is the most time-consuming function in the serial code. This is expected as it is the function that does the most work. We then parallelized the code as described in the *methods* section.

It was then found that the synchronization overhead of the second case for the parallelized code was quite high, and the total time ended up being even slower than the serial code. More specifically, the serial code took **22 ms** on average, while the first parallel code took **33 ms** on average. This was due to the high synchronization overhead.

We then ran the following `perf` command on the parallel code:

```
perf stat -e L1-dcache-loads,L1-dcache-load-misses,l2_rqsts.miss,LLC-loads,
LLC-load-misses ./your_program
```

The results of the `perf` command are as follows:

```
Performance counter stats for './part2-parallel':
    178,913,210      L1-dcache-loads
    10,183,162      L1-dcache-load-misses   #    5.69% of all L1-dcache accesses
    22,011,065      l2_rqsts.miss
    4,080,609      LLC-loads
    481,550      LLC-load-misses           #   11.80% of all LL-cache accesses

    0.076330948 seconds time elapsed

    0.214752000 seconds user
    0.032658000 seconds sys
```

We see that we actually did not get too many cache misses! This suggests our code is not too memory-bound, and that the synchronization overhead was the main bottleneck. However, we have a very high number of cache loads in the first place! This was likely because we are accessing `courses[i].average` repeatedly in the `compute_average` function. This could be optimized by storing the average in a local variable, and then writing it back to the `courses` array at the end.

This was done so, and the second case of the parallel code was also removed, since our data generation never featured a case where the number of courses was less than the number of threads. The optimized parallel code was then run, and the average time was **reduced by 1000%** to **3.33 ms**.