

Project 2 Problem 1

This is the writeup/documentation for Problem 1

Part A - Experiment on own machine

For this part, we can run a simple kernel that repeatedly reads/writes a small array many times. We can increase the array size and see where the **runtime/element** jumps.

Memory Benchmark

The **./memory_benchmark** directory contains files that will perform benchmarks to obtain data for the cache size and performance of the specific computer that the task is running on.

Inside the **./memory_benchmark** directory:

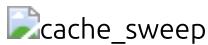
- **mem_bench.c**: a program that repeatedly writes an array, changing the amount of data written each time
- **mem_experiment.sh**: a shell script to automatically run **mem_bench.c** and to collect the necessary data into .csv files
- **plot_all.py**: a python script to plot the data from the corresponding .csv files.

To run the test, just open up a terminal and run:

```
chmod +x ./mem_experiment.sh  
./mem_experiment.sh  
python3 plot_all.py
```

Benchmark Analysis

Below is a plot showing the data from **cache_sweep.csv**



From the plot, we see three distinct drops.

- With block sizes less than 64 KB, we see a bandwidth between 150-160 GB/s. This is the L1 cache performance. The plot shows the L1 cache size to be about 32KB.
- With block sizes between 64KB - 512 KB, we observe a throughput between 130 GB/s to 150 GB/s. We can find the size of the L2 cache by observing where the second plateau ends, which occurs at around 0.5MB. While this doesn't necessarily indicate the size of the L2 cache, it does indicate the effective reuse window to be around 512 KB
- The last plateau is for the L3 cache, which can be observed to have a throughput of 10-15 GB/s, and a working set size of 4-8 MB.

These tests were run on a laptop with an Intel Ultra 7 155H. We can use [Intel's page for the 155H specs](#) to see that our experiments are within reasonable ranges, especially considering that we will be measuring

effective cache usage as opposed to the total cache as reported by Intel.

Part B - ChampSim Simulations

For this part, we install the **ChampSim** simulator. A 1-core and 4-core version of the base architecture is used, and the same instruction window, branch predictor, cache hierarchy, and DRAM model are used. 1 million warmup instructions were used and 10 million simulation instructions were used for the simulations, mainly due to run time length being very long if we scaled up the numbers here.

Two traces were used for evaluation:

- **600.perlbench_s-210B** for a compute-intensive trace with high instruction mix and good locality
- **420.mcf-184B** for a memory-bound trace with pointer-chasing and poor locality

ChampSim Results

To run the benchmarks, there is a **run_champsim_experiments.sh** file located in the **./ChampSim_Runs** directory, with results being output in the **./ChampSim_Runs/results_txt** directory. Below is a table showing the IPC values extracted from ChampSim

Cores	Workload	IPC
1-core	Perlbench	about 1.75-1.9
1-core	MCF	about 0.08
4-core	Perlbench	about 2.1 per core
4-core	MCF	about 0.058 per core

From these results, we can observe that compute workloads scale well with more cores, and memory-bound workloads do not scale well with more cores.

Result Analysis

From the test, Perlbench exhibits:

- High L1 hit rates
- Good spatial locality
- Modest pressure on lower cache levels
- Low DRAM request volume
- Low DRAM buffer miss rates

As a result, Perlbench achieves a high 1-core IPC with good multicore scaling. This relation won't be linear with core count, but we do see this trend due to increased superscalar utilization, effective branch prediction, and low shared-resource contention. Furthermore, the memory latency and bandwidth have minor impacts here because the working sets fit within the L1/L2 caches.

From the test, MCF performs very differently:

- Thrashes private caches
- Dominated by pointer-chasing workloads with poor locality

- Generates enormous numbers of LLC and DRAM misses
- Experiences miss latencies in the hundreds of cycles

This means that adding cores does not scale the IPC, and can even hurt performance due to shared bandwidth and LLC contention. We also observe tens of thousands of row buffer misses, 95% row miss ratio, miss latencies exceeding 200 cycles (up to 800 cycles), and low prefetch usefulness, meaning that the benchmark is completely memory bottlenecked.

Below is a table comparing the results from the two workloads:

Item	Perlbench (Compute)	MCF (Memory-Bound)
Working set	Small/moderate	Large, fragmented
Locality	Good	Very poor
Cache pressure	Low	Severe
DRAM accesses	Low	Extremely high
Sensitivity to latency	Low	Very high
Multicore scaling	Good	Poor
IPC (1-core)	~1.8	~0.08
IPC (4-core)	~2.1/core	~0.058/core