Report

December 25, 2025

Contents

1 Problem 1: Constructing Roofline Model

1.1 Performance Metrics

To create a roofline model, we first needed to get some maximum possible bandwidth estimates for our system. The specs of the system used to run all tests are:

CPU - Ryzen 5 5600H(Zen 3 Ryzen CPU Cores)
Memory - 2 x 8GB 3200 MT/s sticks running in Dual Channel
Operating System - Ubuntu 22.04 LTS

For all our tests for all problems, we used the "nice" command in linux to set the running program to the highest priority so that the operating system prioritizes our program. Also, we used "taskset" command to set CPU affinity of the program under consideration to run on a single logical core. This setup was used in both Problem 1 and Problem 2.

To plot the roofline model, we first needed to calculate max bandwidths of L1, L2, L3 caches and DRAM. We calculate the theoretical maximum bandwidth as follows:

DRAM - Our DRAM can do a maximum of 3200 MT/s. Each transfer is of 8 bytes(the lanes are 64 bit wide) and we are operating the RAM in dual channel configuration. So, we can achieve an effective lane size of 128 bits(16 bytes). Thus max memory transfer speed = 3200 x 16 = 51200 MB/s = 51.2 GB/s
L1 Cache - The L1 cache in a Zen 3 core can transfer 64 bytes of data per CPU clock cycle[4]. The Ryzen 5 5600H CPU is rated at a maximum turbo clock speed of 4.2 GHz. Thus, the max achievable L1 bandwidth = 64 x 4.2 = 268.8GB/s
L2 Cache - The L2 cache in a Zen 3 core can transfer 32 bytes of data per CPU clock cycle[4]. Thus, the max achievable L2 bandwidth = 32 x 4.2 = 134.4GB/s
L3 Cache - The L3 cache in a Zen 3 core can transfer 24 bytes of data per CPU clock cycle[4]. Thus, the max achievable L2 bandwidth = 24 x 4.2 = 100.8GB/s

Now, we calculate the theoretical maximum FLOPS the following way. A Zen 3 AMD core supports 256 bit wide floating point operations. As, we use 64 bit floating point number in Problem 1, we calculate maximum flops for double precision floats. A 256 bit wide amounts to 4 double operations. A Zen 3 core has 2 FADD units and 2 FMA units, each with 256 bit wide registers. An FADD unit contributes one floating point operation per float and each FMA unit contributes two floating point operations per float. This results in a theoretical maximum floating points performance = 4.2(Ghz) x ((2 x 2 x 4) + (4 x 2)) = 100.8 GFLOPS

1.2 Benchmark 1: Matrix-Matrix Multiplication

1.2.1 Implementaion Details

The following versions of matrix multiplication were tested:

- Simple [i,j,k] loop order
- Simple [k,i,j] loop order

- Tiled [i,j,k] loop order
- Tiled [k,i,j] loop order

The source code for the programs can be found at: `https://github.com/I-am-Octavian/HPCA1`

Each program was tested using "perf" to calculate the flops and intensity of the program. To calculate flops, number of floating point instructions retired counter was used. Here, we calculate the effective flops for the program(as some operations might be done speculatively which are not counted here) as it is a more relevant to real world(and also because, there was no counter we could find which could estimate the number of fp instructions issued). Calculating intensity was trickier. For this generation of CPU cores, AMD did not expose any L1 miss counters, or DRAM to cache data movement counters. Neither "perf" nor AMD uProf showed us any availability of such counters. So, we used the l1_data_cache_fills_all counter to count the number of l1 cache fills and then multiplied it by 64(cache line size for a Zen 3 architecture CPU). This gave us a rough estimate of the movement of data to L1 cache from L2 cache and above. At L2 level, only the miss counter was available(no fill counter was available), while at L3, the only counter available was the number of CPU cycles spent to retrieve data to L3 from dram. The following is the command used :

```
sudo nice -n -20 taskset -c 7 perf stat -e "fp_ret_sse_avx_ops.all,
  l1_data_cache_fills_all" ./<matmul_type>
```

1.2.2 Results and Plots

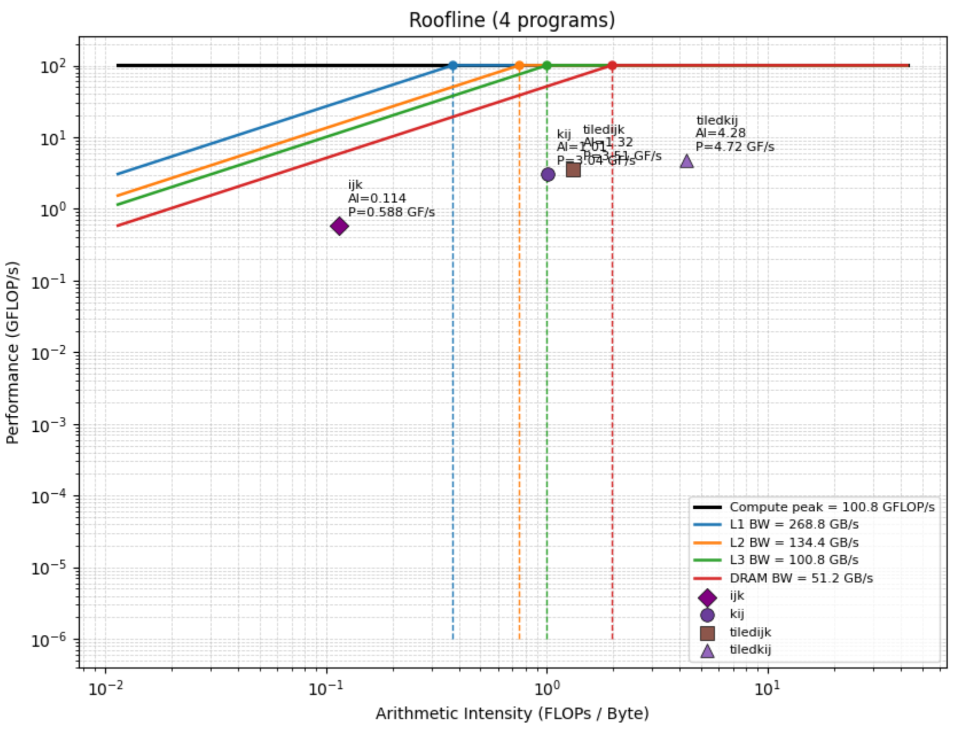

Figure 1: Roofline Model for Matrix Multiplication

1.2.3 Discussion

We can see the ijk matmul is heavily memory bound. It has a very low arithmetic intensity of 0.114 resulting in a low 0.588 GFOPS of performance.

Next we see the kij matmul. It has an AI of 1.01 which is much better than that of ijk version. But is still memeory bound by either L3 or the DRAM

The tiled ijk matmul achieves a slightly better performance then kij matmul. It acheives an AI of 1.32, possibly only memory bound by the DRAM bandwidth

The tiled kij version appears to have the highest AI of 4.28 and as not memory bound but rather compute bound

1.3 Benchmark 2: Breadth First Search (BFS)

1.3.1 Implementation Details

To benchmark our code, we used the same affinity and priority setting described in the previous question.

We compiled the benchmark using clang-14(which in our testing produced much faster binaries than gcc-11). Also, we turned off OpenMP and made sure that the code runs on a single CPU core(same settings as previous question). For all our testing, we generated Kronecker graph with $2\hat{2}5$ vertices. The code modification done to calculate TREPS, is available at : `https://github.com/I-am-Octavian/arch-analysis`

To calculate the bytes moved, we use the same l1 data fills counter as before.

The following command was used:

```
sudo nice -n -20 taskset -c 7 ./bfs -g 25 -n 1
```

1.3.2 Results and Plots

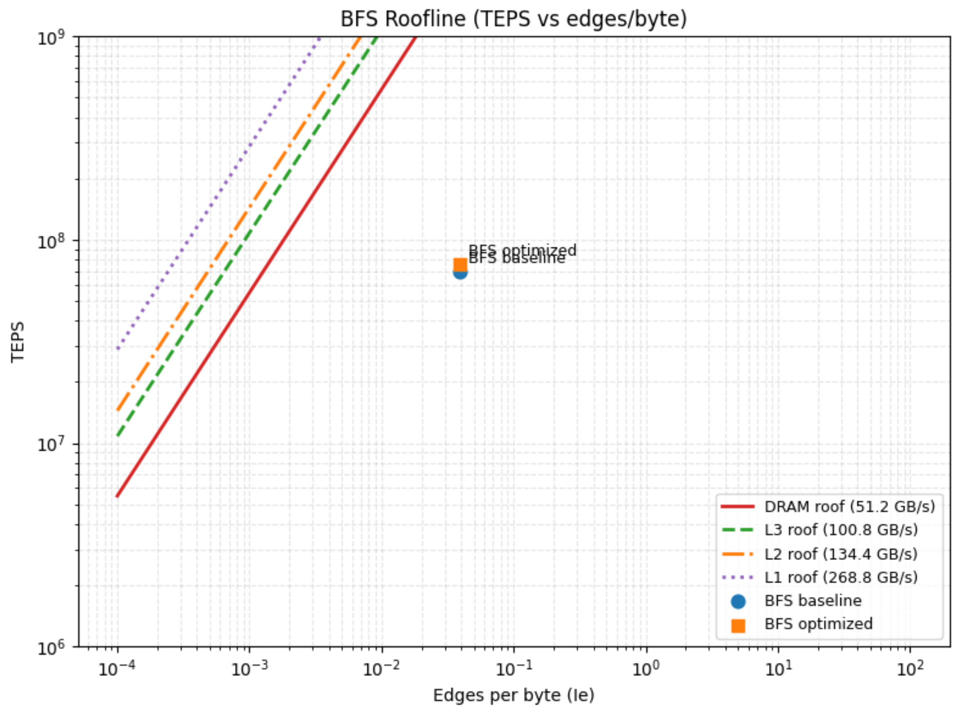

Figure 2: Roofline Model for BFS Computation Phase

1.3.3 Discussion

Usefulness of TREPS: TREPS is a somewhat useful metric. There are no floating point operations as such in a BFS, so calculating TREPS is valid. However, with TREPS, it is not easy to plot a roofline bound for TREPS in a BFS program.

To optimize this, we use a lightweight visited array and check it before reading parent[v] in the top-down step, so most neighbor checks read 1 byte instead of a 4/8-byte parent entry. This aims to reduce bytes per edge and increase the treps. Although, we saw a treps increase from 62e6 to 75e6, there was not much of a change observed in edges per byte. The BFS is still heavily memory bound, which we will discuss later when discussing the CPI stack of the BFS.

2 Problem 2: CPI Stack using Performance Counters and Linear Regression

2.1 Part (a): IPC Characteristics

2.1.1 Methodology

For Problem 2, we choose two benchmarks. The first one was Rodina LU-Decomposition and the second was BFS from GAP. Both benchmarks were run on a single logical core, using the highest priority "niceness"(just as in Problem 1).

To calculate the IPC of the programs, we ran the perf command to record "instructions" and "cycles" counters every 500ms. BFS was run for 30 iterations while the LUD was run for about 230 seconds and then stopped.

The following command was used for BFS:

```
sudo nice -n -20 taskset -c 7 perf stat -e "instructions,cycles" -I
500 --no-big-num -o ipc_data ./bfs -g 25 -n 30
```

And the following command was used for LUD:

```
sudo nice -n -20 taskset -c 7 perf stat -e "instructions,cycles" -I
500 --no-big-num -o ipc_data_lud ./lud_omp -n 1 -s 32000
```

Here, –no-big-num flag is used so that "commas" are not outputted in perf counting values, making parsing the files easier.

The output files were parsed using grep to convert them to a suitable format for plotting

The following is the grep command:

```
grep -Eo "[0-9]\.[0-9][0-9] insn per cycle" ipc_data > ipc_stat_val
```

2.1.2 Results

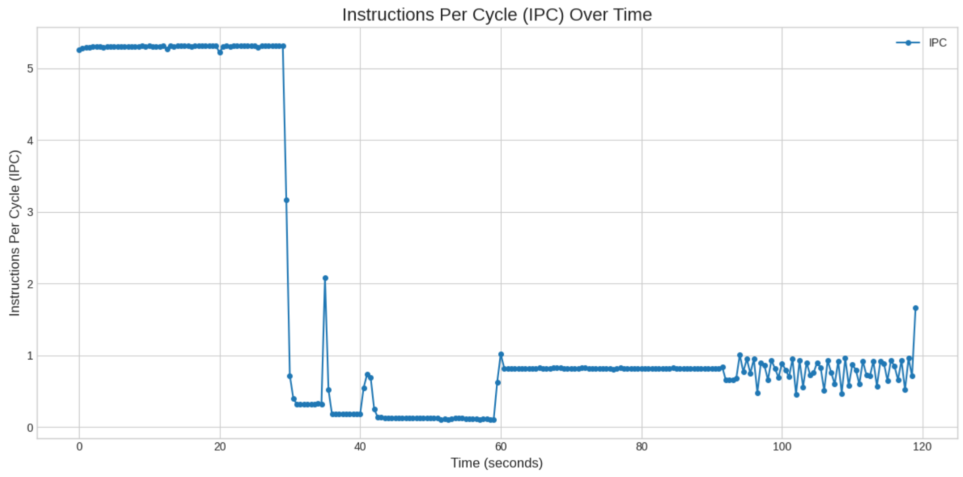

Figure 3: IPC Characteristics Plot(BFS)

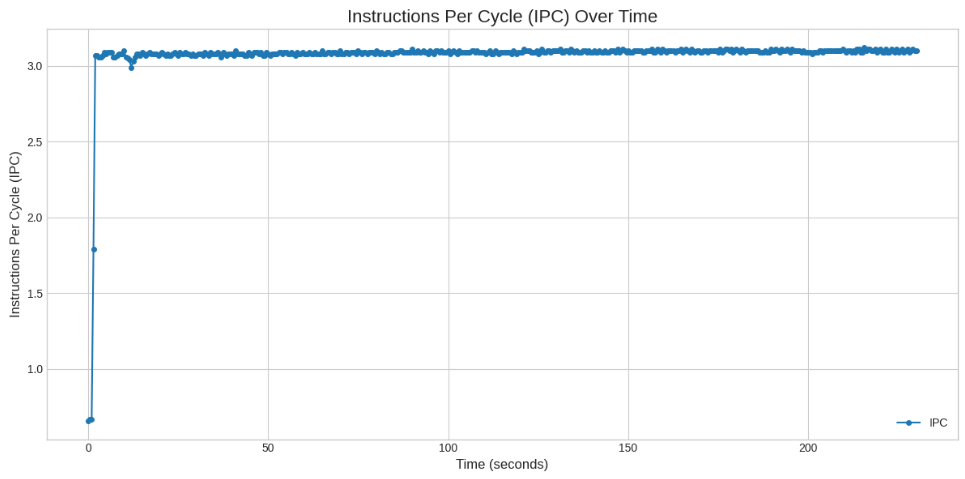

Figure 4: IPC Characteristics Plot (LUD-Rodina)

2.1.3 Discussion

LUD : Its IPC graph is uninteresting. Initially the IPC is low for about 1 second(that is when it is constructing a 32000 x 32000 matrix randomly). Then, when the LUD solver starts, the IPC stays at a consistent about 3 IPC.

BFS: Its IPC graph shows some interesting fluctuations and patterns. There are four region that can be seen from the graph. Let's explore them one by one, starting from the leftmost region. In the first phase, GAP builds a graph till about 30 seconds, which can be observed in the IPC characteristics plot(Figure 3). The next phase, does some precomputation tasks which would eventually make the BFS faster. That phase also lasts for about 30 seconds and has a signifinantly lower IPC(reason will be discussed when discussing the CPI stack). The next two phases are both BFS iterations but with each having a distict characteristic. The "phase 3" is a very "smooth" phase, where the IPC stays fairly constant till about 95 second mark. Then, we observe a "jittery" phase, where the IPC starts to jitter quite a bit. The reason for this "smooth" and then "jittery" phase will be well explained in the next section with the help of the CPI stack.

2.2 Part (b): CPI Stack Construction

2.2.1 Methodology

To construct the CPI stack, the following events were counted:

- instructions
- cycles
- branch-misses
- cache-misses (l2)
- l1_dtlb_misses
- l2_dtlb_misses

- l2_itlb_misses

PS: This makes the number of miss parameters considered 5, which is less than 7, the suggested number of parameters. The number of parameters is restricted to 5 due to our hardware limitations already discusses. AMD Zen 3 CPUs dont expose any l1 cache miss counters, l1_itlb_miss counters or any l3 cache miss counters(as checked by perf list). AMD uProf was unable to provide such counters as well, also suggesting the potential unavailability of such counters.

To construct the data, the data points we collected at an interval of 10ms to create the CPI stack for two programs, LUD from Rohdina and BFS from GAP.

This data was then used to train two regression models to predict the CPI stack for the programs. The data was normalized to "per instruction". Scipy was used to perform Non-Negative Least Squares regression on the dataset generated for the two programs.

The following command was used to generate the data for the regression model.

BFS:

```
sudo nice -n -20 taskset -c 7 perf stat -e "instructions,cycles,
branch-misses, cache-misses,l1_dtlb_misses,l2_dtlb_misses,l2_itlb_misses"
-I 10 --no-big-num -o 2b_data./bfs -g 25 -n 30
```

LUD:

```
sudo nice -n -20 taskset -c 7 perf stat -e "instructions,cycles,
branch-misses, cache-misses,l1_dtlb_misses,l2_dtlb_misses,l2_itlb_misses"
-I 10 --no-big-num -o 2b_data./lud_omp -n 1 -s 32000
```

The following grep command was then used to extract the data:

```
grep -Eo "[0-9]*        (instructions|cycles|branch-misses|cache-
misses|l1_dtlb_misses|l2_dtlb_misses|l2_itlb_misses)" 2b_data >
2b_data_ext
```

2.2.2 Results and Discussion

2.2.3 LUD

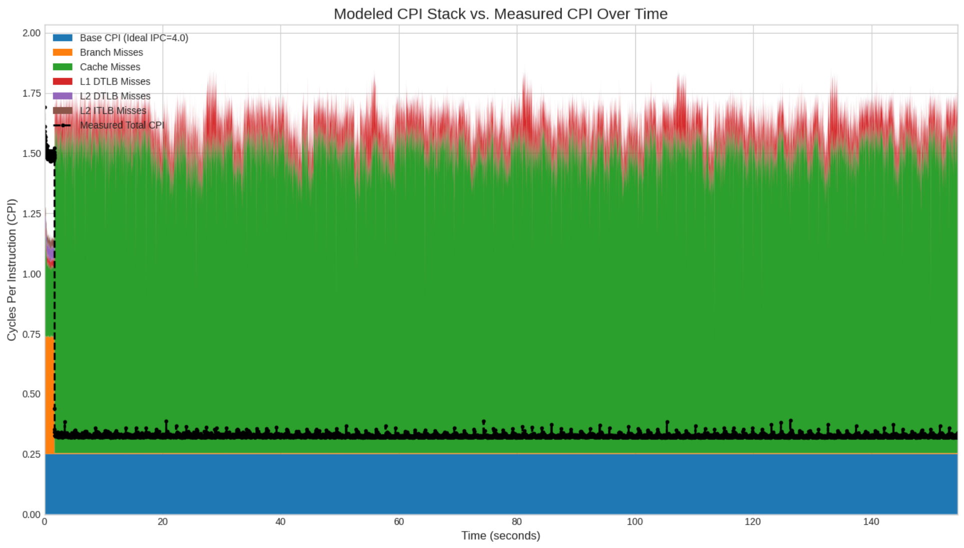

Figure 5: CPI Stack for LUD

We can see from the CPI stack that during the initialization phase of the program, the CPI is high, mainly attributed to the branch misses. After initialization phase is over, the CPI model shows that the CPI is mostly dominated by the L2 cache misses. Crucially, we also plot the actual measured CPI. The difference between the modeled stack and the measured line visually represents the effects of overlap in an out-of-order processor, where the penalties for multiple concurrent miss events are not simply additive.

Next we train a regression model.

The coefficients represent the effective cycle penalty for each miss event.

CPI = 0.3179 (Base CPI) + 0.0000 * (branch_miss_rate) + 0.1265 * (cache_miss_rate) + 0.0000 * (l1_dtlb_miss_rate) + 67.4977 * (l2_dtlb_miss_rate) + 2691.2772 * (l2_itlb_miss_rate)
The model predicts that for LUD, both the branch miss rate and l1 dtlb miss rate play no role and the CPI is more dominated by the l2 performance.
Model Quality

- RMSE: 0.0045
- R-squared: 0.9987
- Adjusted R-squared: 0.9987

Table 1: OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.999
Model:	OLS	**Adj. R-squared:**	0.999
Method:	Least Squares	**F-statistic:**	2.365×10^6
Log-Likelihood:	61684	**Prob (F-statistic):**	0.00
No. Observations:	15462	**AIC:**	-1.234×10^5
Df Residuals:	15456	**BIC:**	-1.233×10^5
Df Model:	5	**Covariance Type:**	nonrobust

2.2.4 BFS

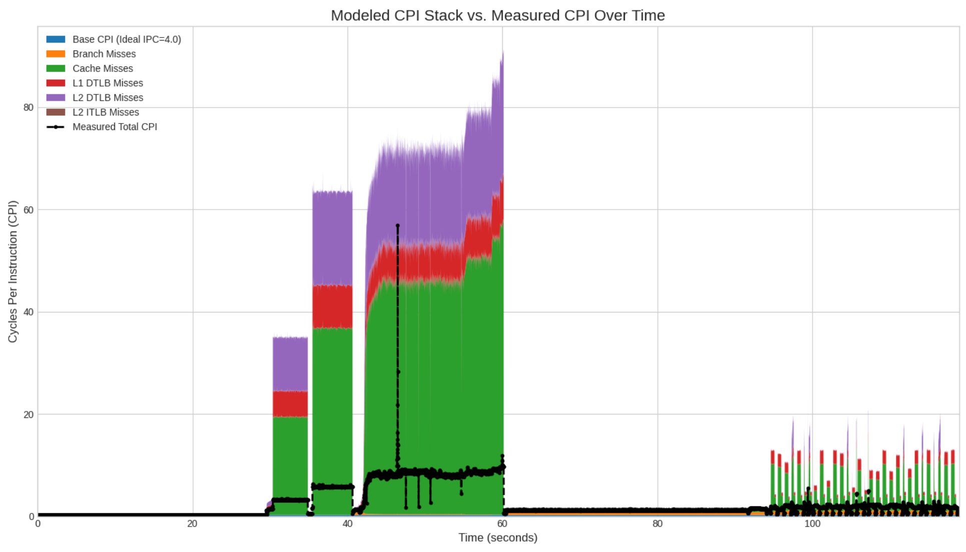

Figure 6: CPI Stack for BFS

We see a similar story for the CPI stack of BFS. The individual misses in an out of order processor are not simply additive and thus produce a much higher CPI than actual. We do note some patterns in the data though, which will be discussed in 2.3

We train a regression model for this as well.

CPI = 0.1320 (Base CPI) + 24.2794 * (branch_miss_rate) + 15.2414 * (cache_miss_rate) + 0.0000 * (l1_dtlb_miss_rate) + 2.8736 * (l2_dtlb_miss_rate) + 2183.7808 * (l2_itlb_miss_rate)

The model predicts that for the CPI of BFS, l1 dtlb misses play no role

Model Quality:

- RMSE: 0.5959
- R-squared: 0.9569
- Adjusted R-squared: 0.9569

Table 2: OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.961
Model:	OLS	**Adj. R-squared:**	0.961
Method:	Least Squares	**F-statistic:**	5.804×10^4
Log-Likelihood:	-10178	**Prob (F-statistic):**	0.00
No. Observations:	11894	**AIC:**	2.037×10^4
Df Residuals:	11888	**BIC:**	2.041×10^4
Df Model:	5	**Covariance Type:**	nonrobust

2.3 Part (c): Phase-wise CPI Stack

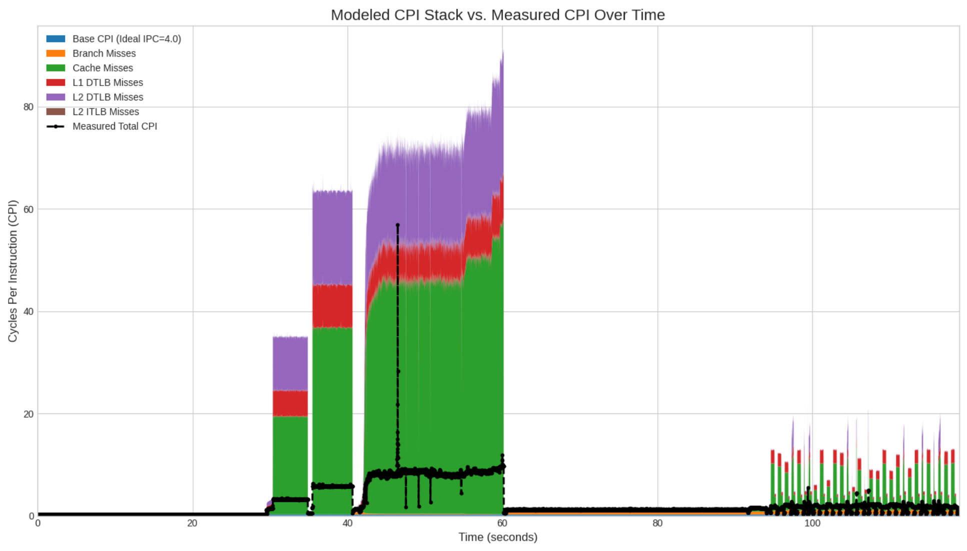

Figure 7: CPI Stacks for BFS

As discussed earlier, there were four distint phases in the BFS which are visible in IPC plot. We now use the CPI stack to make somewhat explain the why the IPC(or CPI) formed four phases. Specifically, we will focus on phases 3 4 which show a different pattern, even though they are both in the execution iteration of BFS.

The phase 1 where the graph is just being initialized is not that interesting. The phase 2, where we are doing some precomputations for BFS, is slower. From the CPI stack, we can see that in this region, there is a very high l2 cache miss rate. This is what we expect as we are creating structures for faster traversal and will encounter a lot of cache misses while initially traversing the graph, while the graph is being loaded into the cache. These precomputations eventually save us a lot of cache misses, as seen from phase 3. Phase 3 is a very smooth, almost constant CPI region, there there are hardly any misses. Then, at around 95 seconds, we start seeing jitter in the CPI graphs. We previously did not explain this pattern as it required looking at the CPI stack. This pattern can now be understood looking at the L2 Cache misses. In earlier iterations, the BFS looks at closer vertices. Those closer vertices are loaded into the caches and thus the ”smooth” phase 3. Then, when the BFS starts to explore the edges farther away, we start encountering more cache misses and the CPU has to wait for the vertices to be loaded into the cache. Thus we see this ”jitter”, which coinsided with increase in L2 cache miss rate.

References

[1] S. Williams et al., "Roofline: An Insightful Visual Performance Model for Multicore Architectures," Communications of the ACM, 2009.

[2] PAPI User's Guide, `http://icl.cs.utk.edu/papi/`.

[3] Linux Kernel Profiling with perf, `https://perf.wiki.kernel.org/index.php/Tutorial`.

[4] AMD Memory Subsystem Architecture `https://chipsandcheese.com/p/amds-zen-4-part-2-memory-subsystem-and-conclusion`