CS341: Computer Architecture Lab

Lab Assignment 4

Report

Devansh Jain (190100044)

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

2021-2022

Contents

Abstract

This lab consists of two parts. In the first part of the lab, we are tasked with profiling and analysing the run-time behaviour of few provided applications using Intel VTune Profiler.
In the second part of the lab, we are tasked with running the same applications on a simulator, known as `ChampSim`, to understand the performance impact caused by different configurations of caches in a system.

From second pat of the lab, we can conclude the provided programs weren't affected by change in MSHR size - IPC remained constant to 4th decimal place.
We saw a significant increase in misses when using direct mapped cache - MPKI for L1 cache increased by 300% for matrix multiplication programs, 950% for bfs program and 1800% for quicksort program.
As we see no effect of increasing the associativity, we can safely conclude that the current level of associativity is good for the given programs.
Changing the cache size only has positive effect on bfs program (increasing cache size reduces MPKI for L1D by 30%).

Part 0: Getting Things Ready

Install `Intel VTune Profiler`

Installed successfully using the stand-alone app using offline installer script present in this link.

It was pretty easy to install `VTune` using the script.
While installing it showed that I didn't have `XCB` and `DRM` packages installed. Upon checking, I confirmed that they were already present.
Even though it failed prerequisites, there was a next option. I didn't face any issues for the rest of installation process.
From start to end, it took around 10-12 minutes to have the application installed, followed by 3-5 minutes for tutorial.

Install `Docker`

Had docker setup from other projects.
Version: 20.10.9

Pull `ChampSim` Image

Pulled `0xd3ba/champsim-lab:latest`

Part 1: Profiling with VTune

1.1 `bfs.cpp`

Performance Snapshot

- IPC: 1.830
- Logical Core Utilization: 8.2% (0.979 out of 12)
- Physical Core Utilization: 16.2% (0.973 of 6)
- Memory bound: 32.0% of Pipeline slots

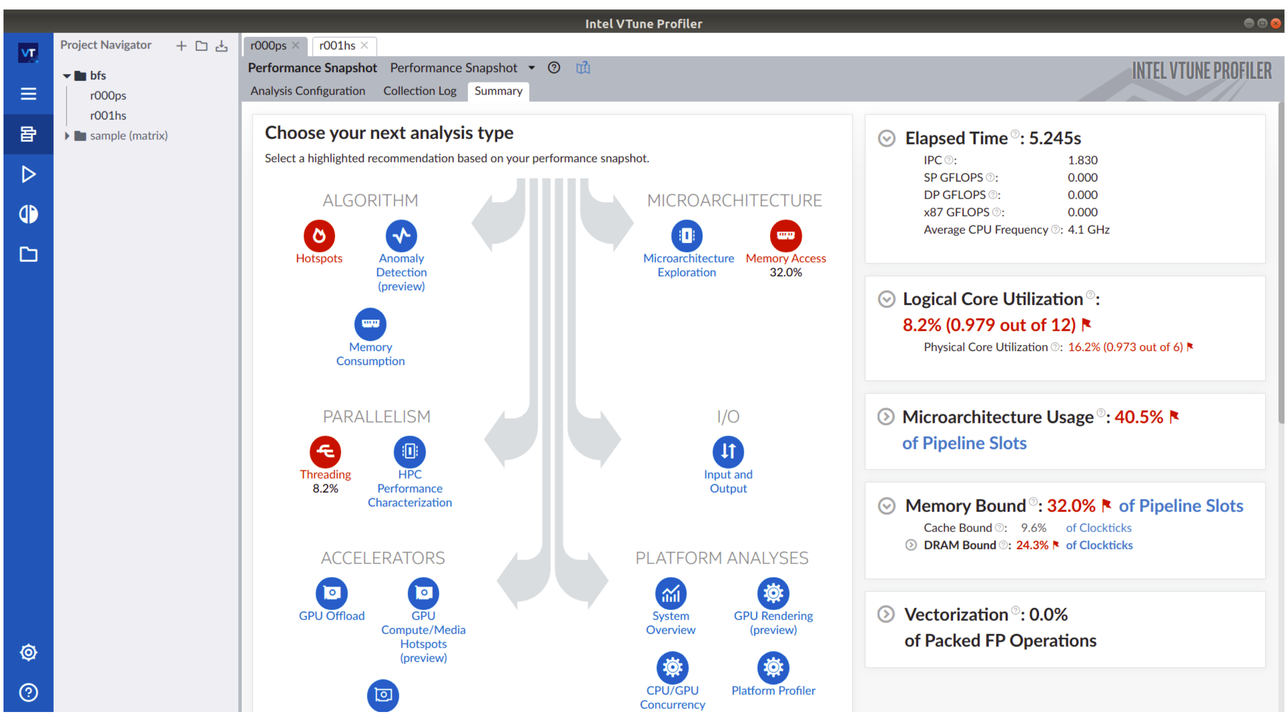

Figure 1.1: Performance Snapshot for `bfs.cpp`

Top 5 Functions by CPU Time

Function	Module	CPU Time
bfs	bfs.o	2.621s
main	bfs.o	1.156s
_int_free	libc-2.27.so	0.236s
_int_malloc	libc-2.27.so	0.154s
__gnu_cxx::new_allocator<Node*>::construct<Node*, Node* const&>	bfs.o	0.124s

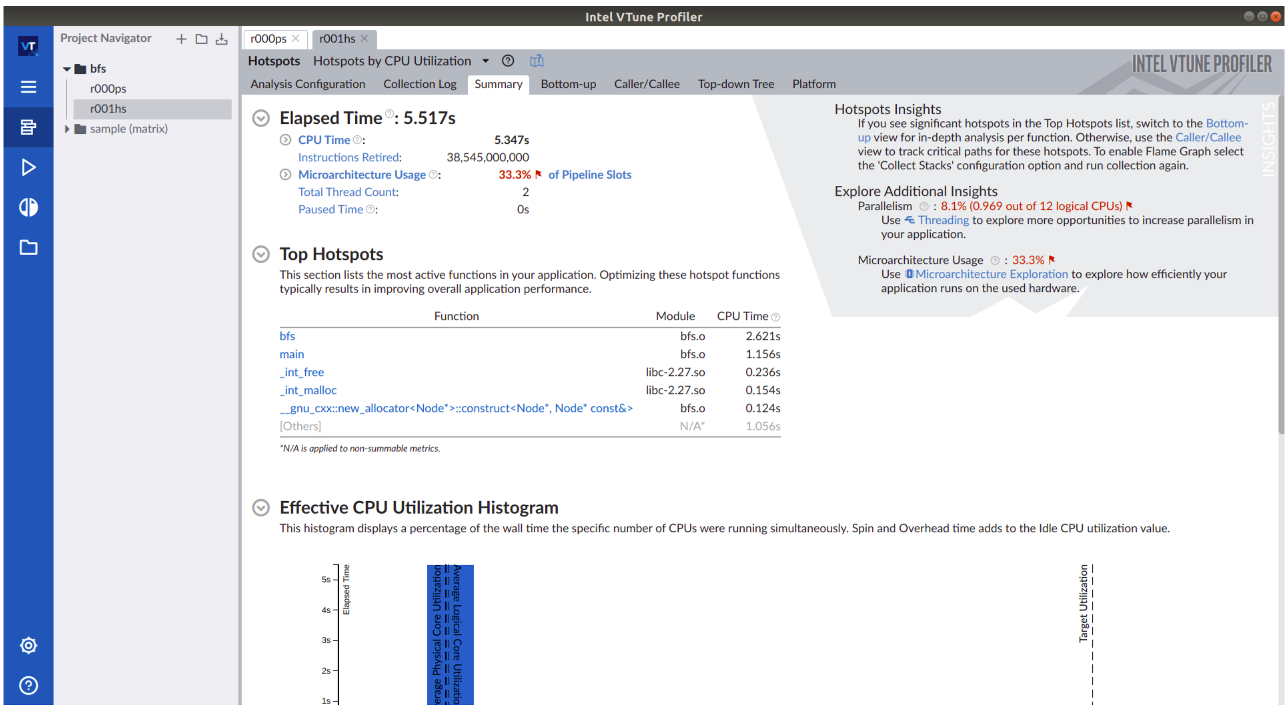

Figure 1.2: Top Functions by CPU Time for `bfs.cpp`

Top 5 Source lines by CPU Utilization

Source	Function	CPU Utilization
`if (left_child) node_Q.push(left_child);`	`inline void bfs(Node *root)`	22.7%
`bfs(root);`	`int main()`	21.6%
`right_child = curr_node->right;`	`inline void bfs(Node *root)`	17.2%
`for (int i = 0; i < q_size; i++) {`	`inline void bfs(Node *root)`	4.8%
`left_child = curr_node->left;`	`inline void bfs(Node *root)`	3.2%

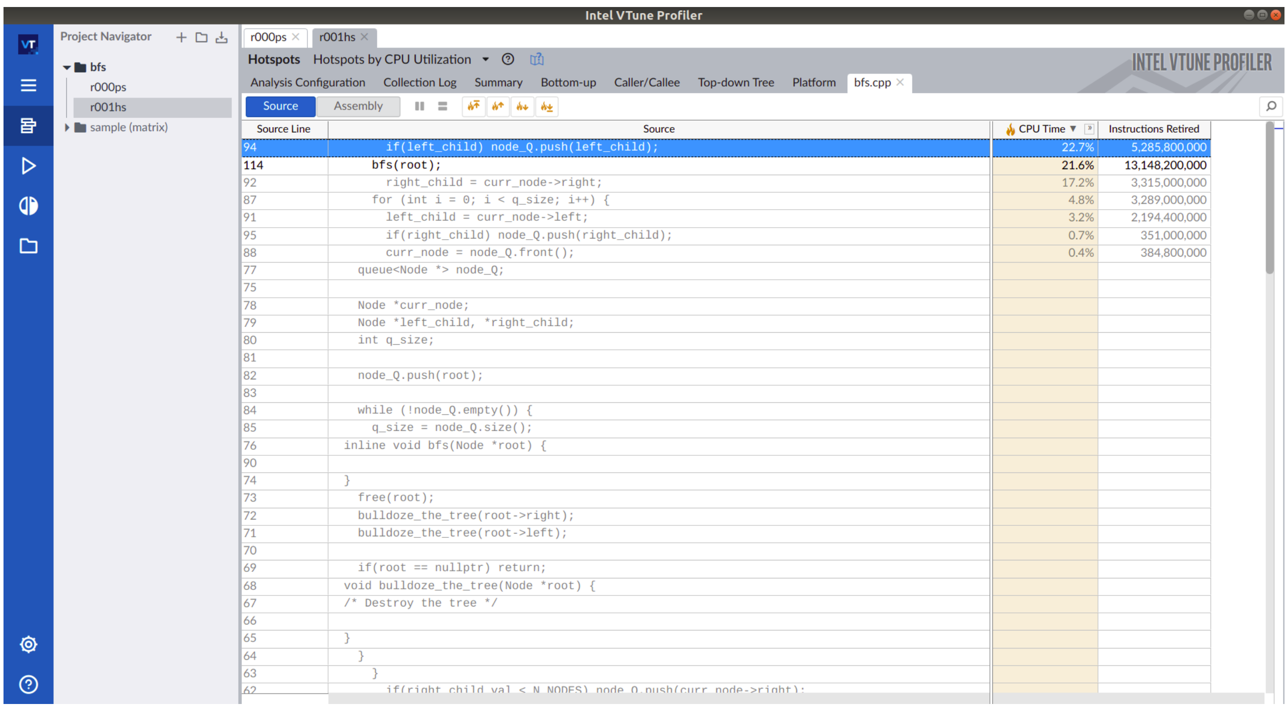

Figure 1.3: Top Source lines by CPU Utilization for `bfs.cpp`

Inference

We see that majority of the time goes in function `bfs`.

The time consuming line in `main` is calling `bfs(root)`. This would be due to the need to set the function stack and as it is done for `N_LOOPS` times, it climbs above `plant_a_tree` and `bulldoze_the_tree` stack setup.

Every line in `bfs` is called repeatedly due to 3 level of loops (2 level in `bfs` and 1 level in `main`). These lines combined consume the majority of CPU time (~50%).

1.2 `matrix_multi.cpp`

Performance Snapshot

- IPC: 0.874
- Logical Core Utilization: 8.2% (0.982 out of 12)
- Physical Core Utilization: 16.3% (0.976 of 6)
- Memory bound: 59.1% of Pipeline slots

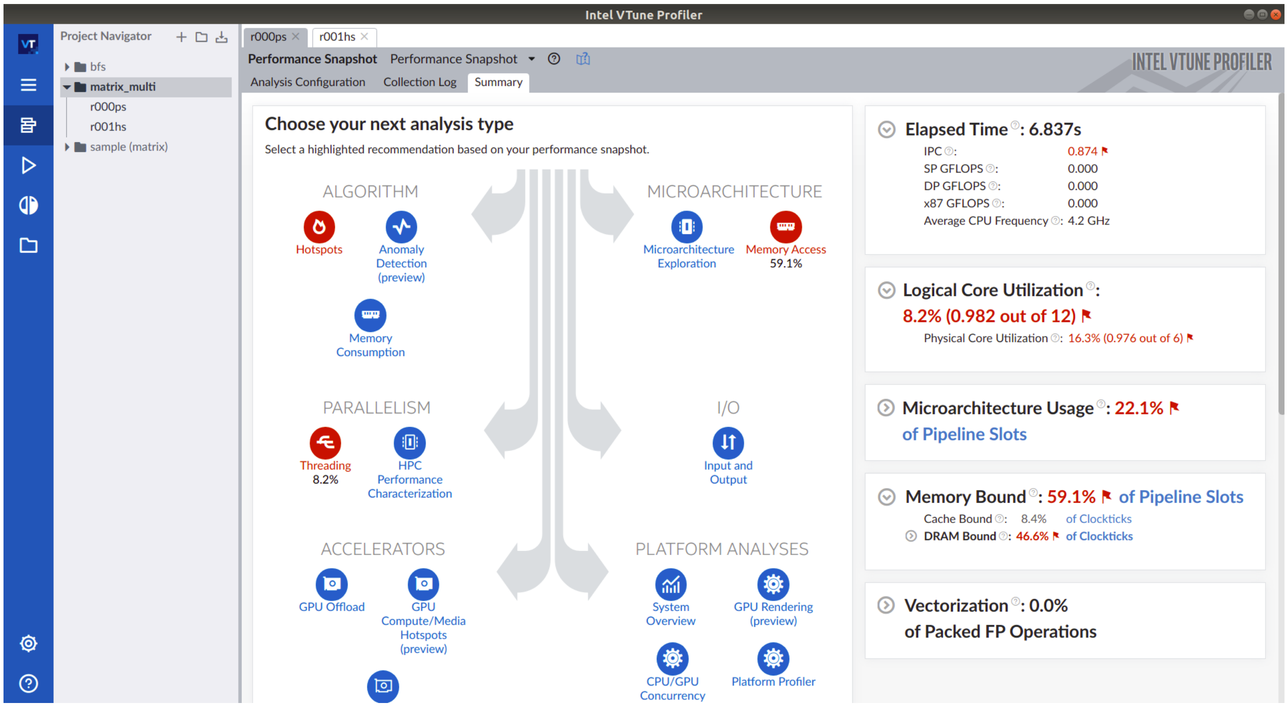

Figure 1.4: Performance Snapshot for `matrix_multi.cpp`

Top Functions by CPU Time

Function	Module	CPU Time
`matrix_product`	`matrix_multi.o`	6.597s

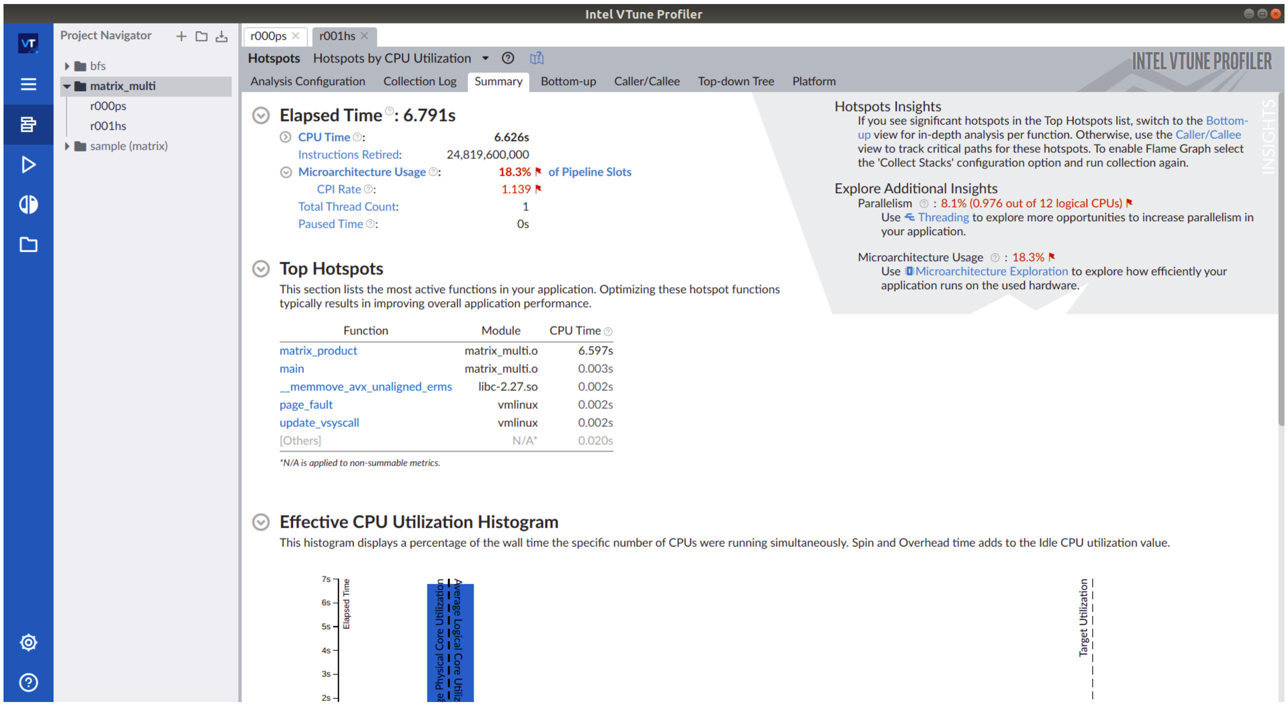

Figure 1.5: Top Functions by CPU Time for `matrix_multi.cpp`

Top 2 Source lines by CPU Utilization

Source	Function	CPU Utilization
`C[i][j] += A[i][k] * B[k][j];`	`void matrix_product()`	90.9%
`for (int k = 0; k < N_DIMS; k++) {`	`void matrix_product()`	8.6%

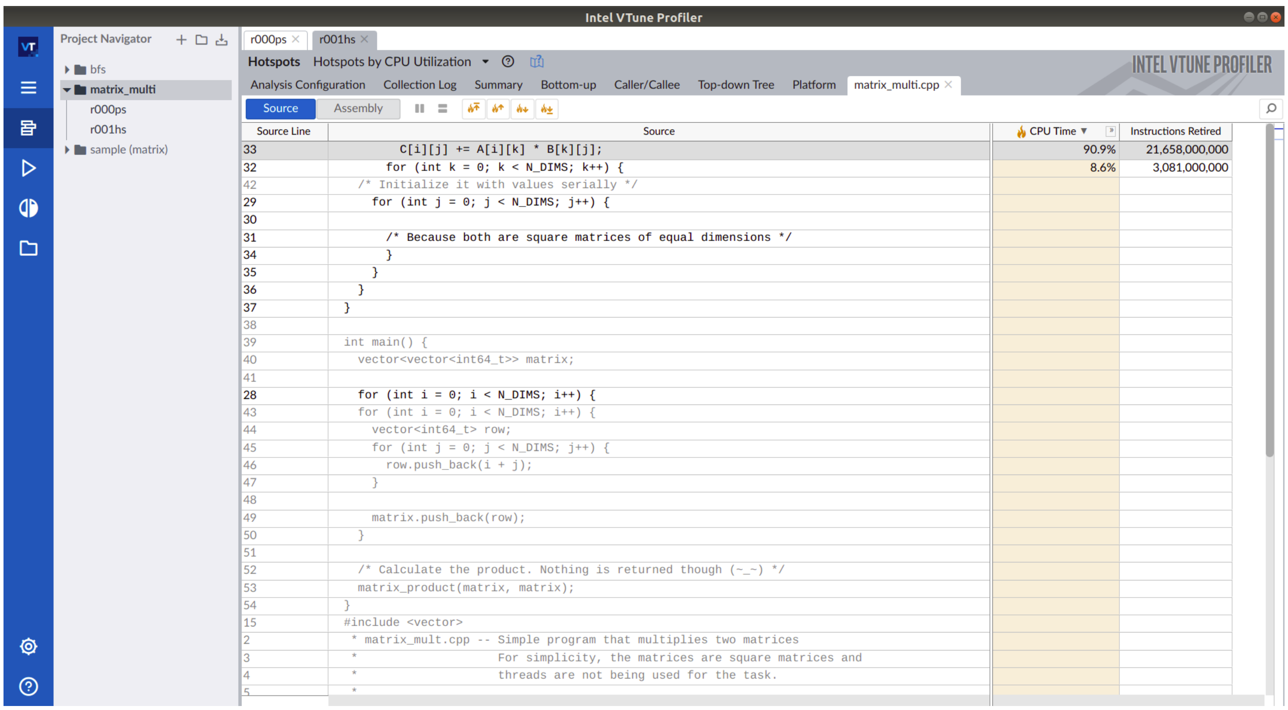

Figure 1.6: Top Source lines by CPU Utilization for `matrix_multi.cpp`

Inference

We see that majority of the time goes in function `matrix_product`.

As discussed in lectures, `ijk` is not the optimal loop order for memory access. We see that the inner loop takes most of the time.
And we access and modify `k` at every iteration of the inner loop, it is the line to take second most CPU time.

1.3 matrix_multi_2.cpp

Performance Snapshot

- IPC: 1.339
- Logical Core Utilization: 8.2% (0.981 out of 12)
- Physical Core Utilization: 16.2% (0.973 of 6)
- Memory bound: 38.0% of Pipeline slots

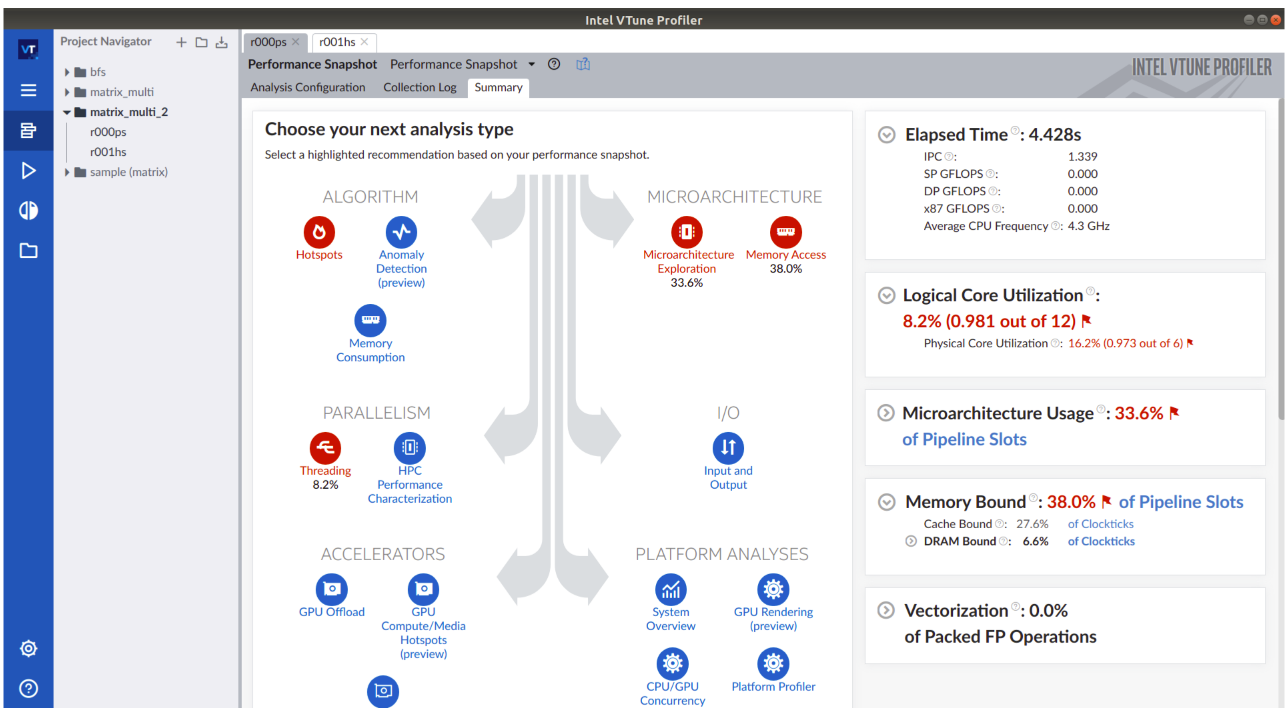

Figure 1.7: Performance Snapshot for `matrix_multi_2.cpp`

Top Functions by CPU Time

Function	Module	CPU Time
matrix_product	matrix_multi_2.o	4.492s

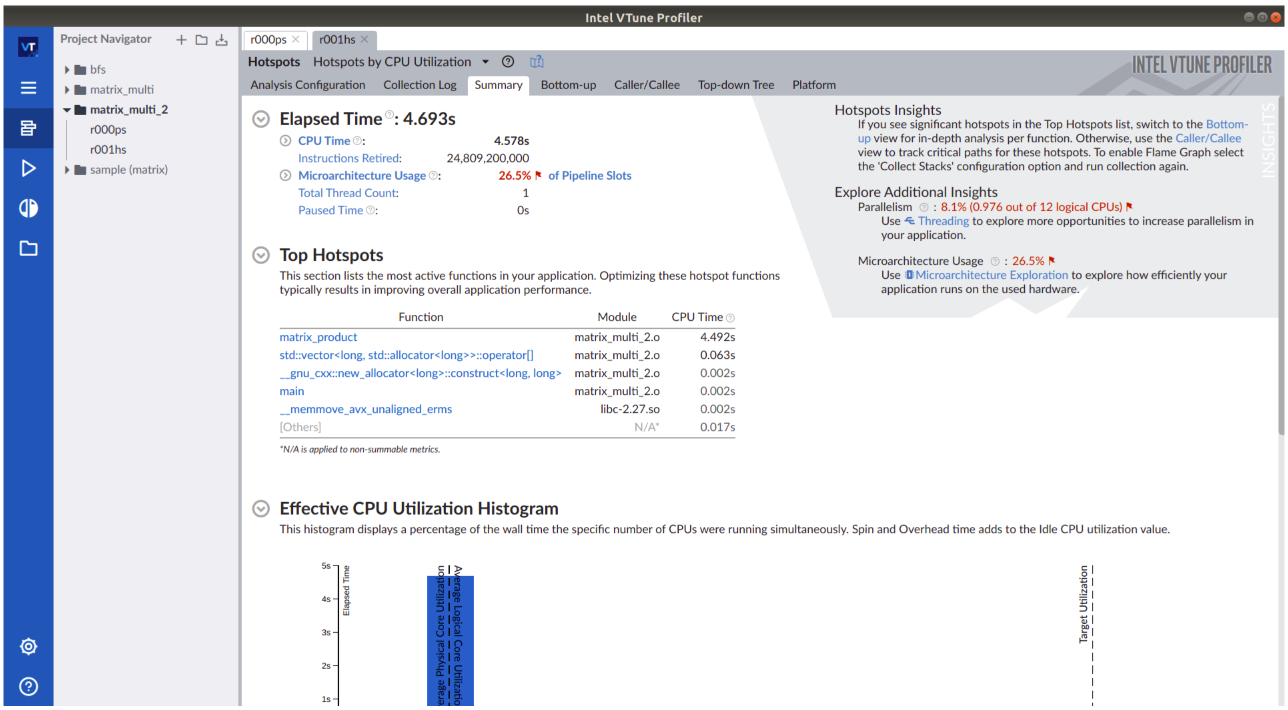

Figure 1.8: Top Functions by CPU Time for `matrix_multi_2.cpp`

Top 2 Source lines by CPU Utilization

Source	Function	CPU Utilization
`C[i][j] += A[i][k] * B[k][j];`	`void matrix_product()`	84.4%
`for (int k = 0; k < N_DIMS; k++) {`	`void matrix_product()`	13.7%

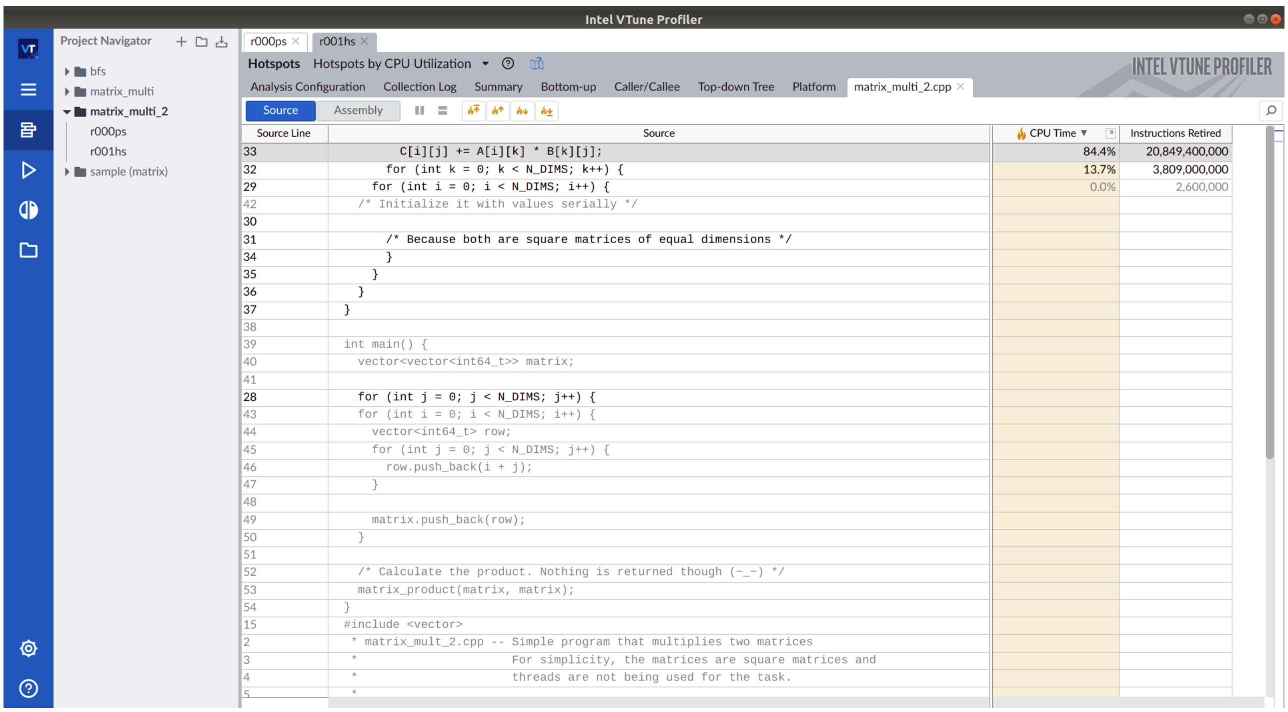

Figure 1.9: Top Source lines by CPU Utilization for `matrix_multi_2.cpp`

Inference

We see that majority of the time goes in function `matrix_product`.

As discussed in lectures, `jik` (same as `ijk`) is not the optimal loop order for memory access. We see that the inner loop takes most of the time.
And we access and modify `k` at every iteration of the inner loop, it is the line to take second most CPU time.

1.4 quicksort.cpp

Performance Snapshot

- IPC: 0.748
- Logical Core Utilization: 8.0% (0.966 out of 12)
- Physical Core Utilization: 15.7% (0.941 of 6)
- Memory bound: 23.0% of Pipeline slots

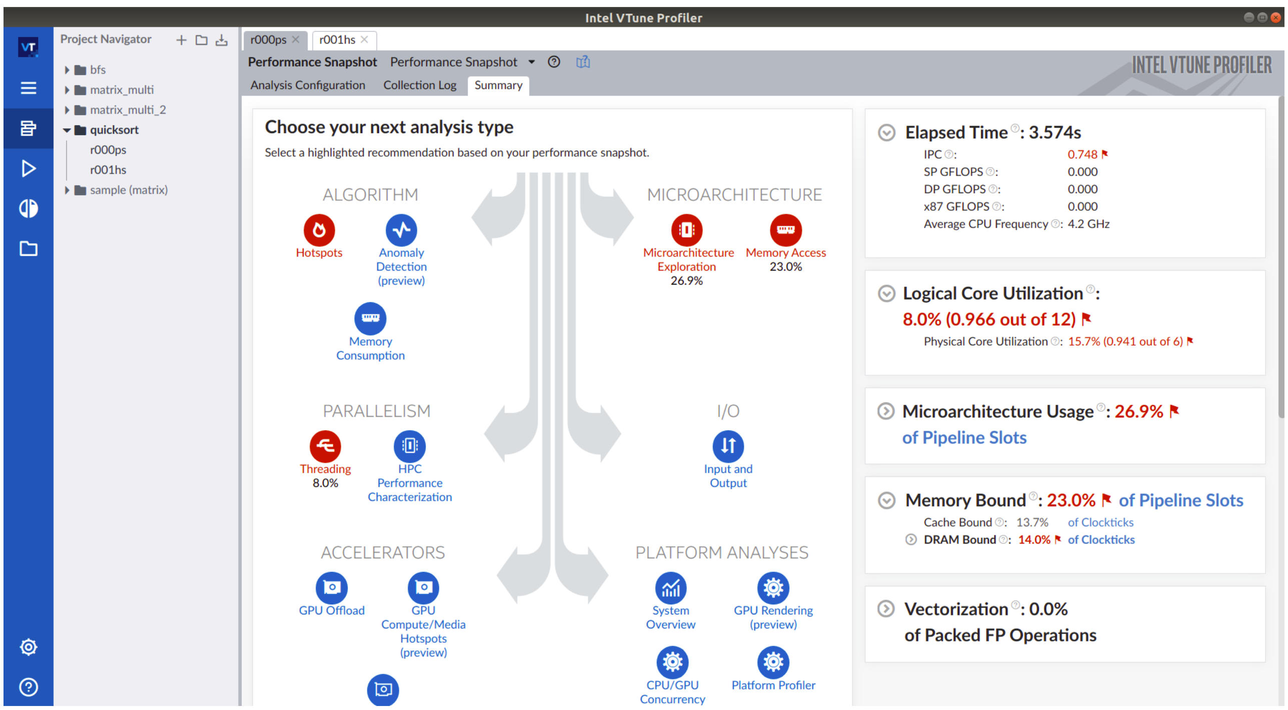

Figure 1.10: Performance Snapshot for `quicksort.cpp`

Top Functions by CPU Time

Function	Module	CPU Time
`__memmove_avx_unaligned_erms`	`libc-2.27.so`	0.875s
`page_fault`	`vmlinux`	0.511s
`clear_page_erms`	`vmlinux`	0.228s
`prepare_exit_to_usermode`	`vmlinux`	0.226s
`perf_iterate_ctx`	`vmlinux`	0.155s
Others	N/A	1.545s

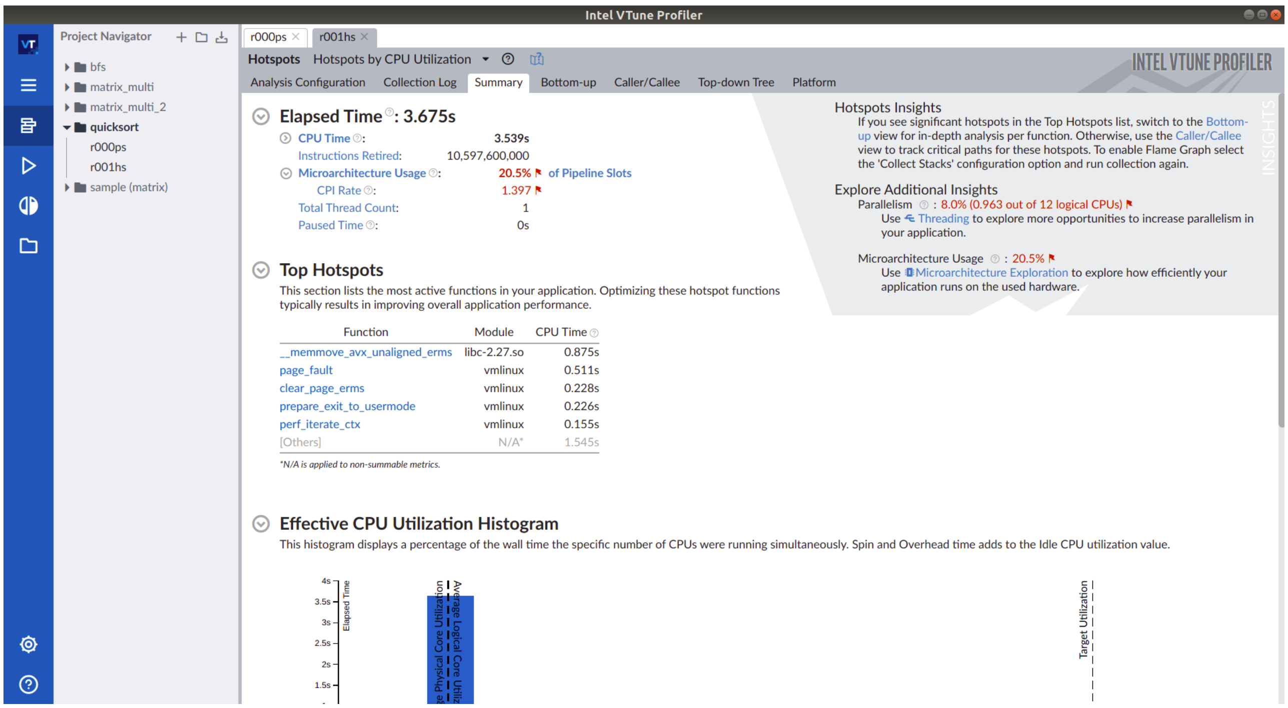

Figure 1.11: Top Functions by CPU Time for `quicksort.cpp`

Top 5 Source lines by CPU Utilization

Source	Function	CPU Utilization
`b = c;`	`void swap()`	2.1%
`if (nums[i] < pivot) {`	`long partition()`	1.9%
`slow_ptr++;`	`long partition()`	1.7%
`for (long i = lo; i < hi; i++) {`	`long partition()`	0.6%
`a = b;`	`void swap()`	0.2%

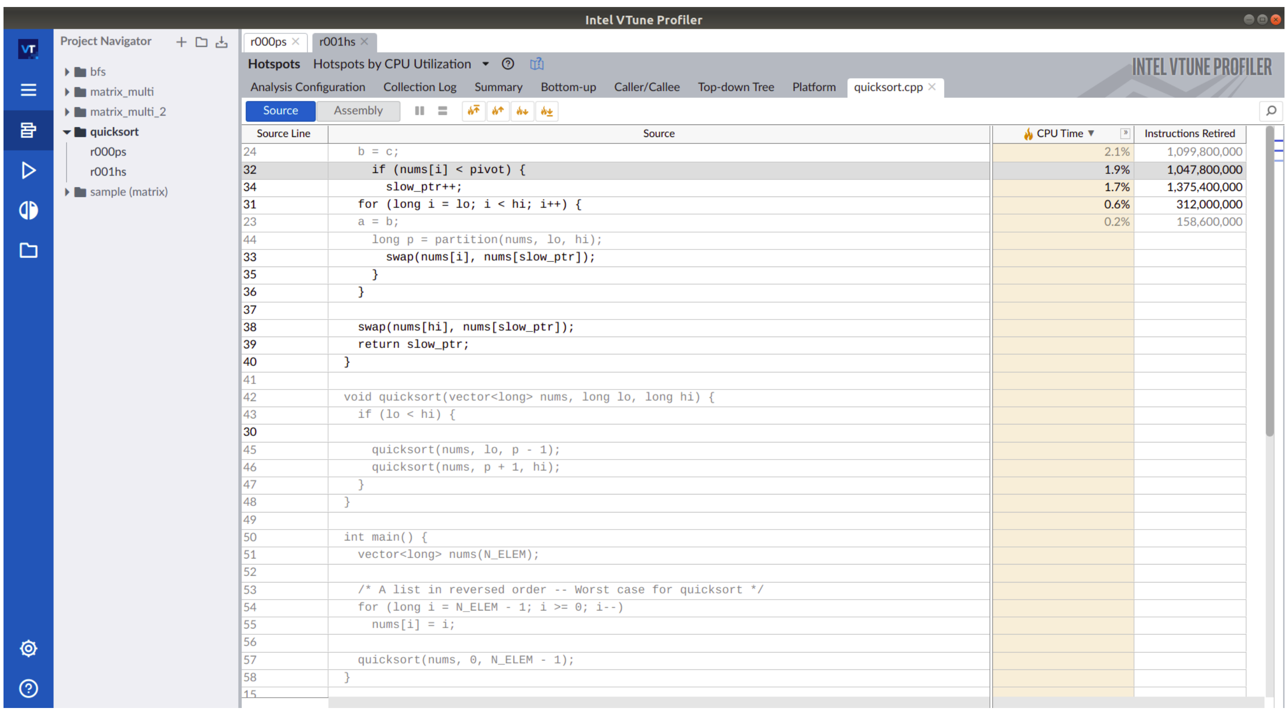

Figure 1.12: Top Source lines by CPU Utilization for `quicksort.cpp`

Inference

We see that majority of the time goes in handling page faults and `memmove`. Possible explanation is that because it crosses my limit of RAM and overflows in swap memory, we might be getting page faults and page needs to be loaded back from the swap memory. (We discussed this in OS course)

The lines (present in `quicksort.cpp`) consuming the majority of time is mostly because of the number of times it is executed.

Quicksort algorithm is mostly partitioning and swapping, so those two functions take the majority of the time.

Part 2: Simulating with ChampSim

2.1 Prepare traces

Generate tracer for champsim:

```
cd /champsim/ChampSim/tracer; ./make_tracer.sh;
```

Used pin to generate traces:

```
pin -t obj-intel64/champsim_tracer.so -o <program>.trace -t 21000000 - <executable>;
```

Used xz to compress the traces:

```
xz -vz <program>.trace -threads=0;
```

Program	Parameters	Execution time	Trace size
bfs.o	N_NODES (1«15); N_LOOPS 1000;	3.4 s	2092 KB
matrix_multi.o	N_DIMS 700;	4.5 s	2172 KB
matrix_multi_2.o	N_DIMS 700;	4.2 s	2160 KB
quicksort.o	N_ELEM (1«14);	3.1 s	4172 KB

2.2 Setup Configurations

To speed up the task and to avoid having to reset to default values again and again, I prepared 7 copies of ChampSim in the docker container.
This helped me run the 28 simulations in parallel and the entire experiment took 5-6 minutes to finish.

To setup each of the Configurations, I had to modify `./inc/cache.h`.

I have used CACTI to compute the latency updates for changes in cache size.
Theoretically, there would be change in latency when we change associativity as well but as it wasn't present in PS I have ignored it.
As hinted by professor, latency of 12-way cache is computed by taking mean of latency of 8-way and 16-way caches keeping sets as constant.
For `L1I` and `L1D` caches, the change in access time was small, so the latency remains same across the configurations.
Also, I have rounded off the latency to nearest integers to avoid issues with ChampSim.
(For example, I got `L1I_LATENCY` to be 3.8 and 4.2 for half and double size cache respectively. I have consider both of them as 4 - same as baseline)

Updates in `./inc/cache.h`

Line	Parameter	Baseline	Direct Mapped	Fully Associative	Reduced Size	Doubled Size	Reduced MSHR	Doubled MSHR
46	`L1I_SET`	64	64*8	1	32	128	64	64
47	`L1I_WAY`	8	1	8*64	8	8	8	8
51	`L1I_MSHR_SIZE`	8	8	8	8	8	4	16
52	`L1I_LATENCY`	4	4	4	4	4	4	4
55	`L1D_SET`	64	64*12	1	32	128	64	64
56	`L1D_WAY`	12	1	12*64	12	12	12	12
60	`L1D_MSHR_SIZE`	16	16	16	16	16	8	32
61	`L1D_LATENCY`	5	5	5	5	5	5	5
64	`L2C_SET`	1024	1024*8	1	512	2048	1024	1024
65	`L2C_WAY`	8	1	8*1024	8	8	8	8
69	`L2C_MSHR_SIZE`	32	32	32	32	32	16	64
70	`L2C_LATENCY`	10	10	10	9	13	10	10
73	`LLC_SET`	2048	2048*16	1	1024	4096	2048	2048
74	`LLC_WAY`	16	1	16*2048	16	16	16	16
78	`LLC_MSHR_SIZE`	64	64	64	64	64	32	128
79	`LLC_LATENCY`	20	20	20	17	24	20	20

2.3 Run simulations

Build champsim:

```
./build_champsim.sh bimodal no no no no lru 1;
```

Run simulation for traces:

```
./run_champsim.sh bimodal-no-no-no-no-lru-1core 10 10 bfs.trace.xz &;
./run_champsim.sh bimodal-no-no-no-no-lru-1core 10 10 matrix_multi.trace.xz &;
./run_champsim.sh bimodal-no-no-no-no-lru-1core 10 10 matrix_multi_2.trace.xz &;
./run_champsim.sh bimodal-no-no-no-no-lru-1core 10 10 quicksort.trace.xz &;
```

We can run all 28 simulations in the background and watch the processes using `watch -n 0.5 ps;`. The entire experiment ends in around 10 minutes.

2.3.1 `bfs.trace.xz`

Parameter		Baseline	Direct Mapped	Fully Associative	Reduced Size	Doubled Size	Reduced MSHR	Doubled MSHR
cycles		11505060	11548164	11510528	11646705	11410653	11505205	11504968
IPC		0.869183	0.865938	0.86877	0.858612	0.876374	0.869172	0.86919
Cache Misses	`L1D`	21942	30732	21942	21947	21931	21942	21972
	`L1I`	1	201163	1	9	1	1	1
	`L2C`	21464	23645	21727	21881	14648	21464	21464
	`LLC`	13437	14032	13437	14920	13436	13437	13437
MPKI	`L1D`	2.1942	3.0732	2.1942	2.1947	2.1931	2.1942	2.1942
	`L1I`	0.0001	20.1163	0.0001	0.0009	0.0001	0.0001	0.0001
	`L2C`	2.1464	2.3645	2.1727	2.1881	1.4648	2.1464	2.1464
	`LLC`	1.3437	1.4032	1.3437	1.4920	1.3436	1.3437	1.3437
Avg Miss Latency	`L1D`	91.8093	71.4266	91.6041	130.174	85.5957	91.7903	91.844
	`L1I`	215	14.1125	215	57.6667	229	215	215
	`L2C`	78.5156	73.5786	77.3577	116.525	107.193	78.4964	78.5508
	`LLC`	77.5087	79.0059	76.5858	132.767	76.5338	77.4779	77.5649

2.3.2 `matrix_multi.trace.xz`

Parameter		Baseline	Direct Mapped	Fully Associative	Reduced Size	Doubled Size	Reduced MSHR	Doubled MSHR
cycles		17624045	17696897	17624045	17625492	17624129	17624452	17623986
IPC		0.567407	0.565071	0.567407	0.56736	0.567404	0.567394	0.567409
Cache Misses	`L1D`	7392	20961	7393	9130	7381	7392	7392
	`L1I`	0	8396	0	174	0	0	0
	`L2C`	7269	7707	7269	7481	7268	7269	7269
	`LLC`	7268	7548	7268	7268	7268	7268	7268
MPKI	`L1D`	0.7392	2.0961	0.7393	0.9130	0.7381	0.7392	0.7392
	`L1I`	0.0000	0.8396	0.0000	0.0174	0.0000	0.0000	0.0000
	`L2C`	0.7269	0.7707	0.7269	0.7481	0.7268	0.7269	0.7269
	`LLC`	0.7268	0.7548	0.7268	0.7268	0.7268	0.7268	0.7268
Avg Miss Latency	`L1D`	117.04	52.756	117.026	101.301	130.94	116.558	118.607
	`L1I`	-	14.2587	-	20.1954	-	-	-
	`L2C`	103.766	103.328	103.766	106.711	114.666	103.277	105.36
	`LLC`	73.7643	76.0135	73.7643	83.4732	77.6564	73.275	75.3581

2.3.3 `matrix_multi_2.trace.xz`

Parameter		Baseline	Direct Mapped	Fully Associative	Reduced Size	Doubled Size	Reduced MSHR	Doubled MSHR
cycles		17622240	17694653	17622240	17623493	17622347	17622647	17622268
IPC		0.567465	0.565143	0.567465	0.567424	0.567461	0.567452	0.567464
Cache Misses	`L1D`	7392	21323	7393	9140	7381	7392	7392
	`L1I`	0	6867	0	174	0	0	0
	`L2C`	7269	7857	7269	7483	7268	7269	7269
	`LLC`	7268	7465	7268	7268	7268	7268	7268
MPKI	`L1D`	0.7392	2.1323	0.7393	0.9140	0.7381	0.7392	0.7392
	`L1I`	0.0000	0.6867	0.0000	0.0174	0.0000	0.0000	0.0000
	`L2C`	0.7269	0.7857	0.7269	0.7483	0.7268	0.7269	0.7269
	`LLC`	0.7268	0.7645	0.7268	0.7268	0.7268	0.7268	0.7268
Avg Miss Latency	`L1D`	116.464	53.2717	116.329	101.704	130.203	116.168	117.712
	`L1I`	-	14.8274	-	19.4425	-	-	-
	`L2C`	103.181	105.285	103.057	107.273	113.948	102.88	104.45
	`LLC`	73.1791	81.0347	73.0553	84.0922	76.9375	72.8782	74.4483

2.3.4 `quicksort.trace.xz`

Parameter		Baseline	Direct Mapped	Fully Associative	Reduced Size	Doubled Size	Reduced MSHR	Doubled MSHR
cycles		23473142	32062866	23473142	23492281	23585074	23473142	23473142
IPC		0.426019	0.311887	0.426019	0.425672	0.423997	0.426019	0.426019
Cache Misses	`L1D`	62136	1191142	62136	62136	62136	62136	62136
	`L1I`	0	102	0	0	0	0	0
	`L2C`	12300	16609	12300	14630	12297	12300	12300
	`LLC`	12296	18305	12293	12297	12290	12296	12296
MPKI	`L1D`	6.2136	119.1142	6.2136	6.2136	6.2136	6.2136	6.2136
	`L1I`	0.0000	0.0102	0.0000	0.0000	0.0000	0.0000	0.0000
	`L2C`	1.2300	1.6609	1.2300	1.4630	1.2297	1.2300	1.2300
	`LLC`	1.2296	1.8305	1.2293	1.2297	1.2290	1.2296	1.2296
Avg Miss Latency	`L1D`	37.1162	11.9622	37.2176	52.8647	40.495	37.1162	37.1162
	`L1I`	-	90.6176	-	-	-	-	-
	`L2C`	111.698	119.162	112.211	165.043	113.641	111.698	111.698
	`LLC`	81.725	81.2332	82.2574	165.422	76.6843	81.725	81.725

2.4 Results

2.4.1 Direct Mapped

IPC decreases significantly for direct-mapped configuration - approx 27% for quicksort and 0.5% for others. The reason is that due to reduction in associativity, the number of collision misses increase. And as quicksort revisits the same address a lot of times ($\mathcal{O}(N)$), the decrease is much more significant.

It highly affects the MPKI of L1 caches. Due to smaller size, the chances of collisions increase and it explodes the change in MPKI (had to trim the plot due to this fact). It is 4x for matrix multiplication programs, 10.5x for bfs and 19x for quicksort.

This effect is reduced and carried over to the lower caches.

Unlike baseline where most of the misses had to go to the DRAM, here a lot of the misses are found in the next level cache. This reduces the Average Miss Latency.

2.4.2 Fully Associative

IPC remains constant for fully-associative. The number of collision misses in baseline would be close to zero, thus increasing associativity doesn't affect the IPC either.

MPKI and Average Miss latency also remains constant (up to a few decimal places)

2.4.3 Reduced Size

IPC isn't affected by changing cache size in matrix multiplication programs. The reason is the working set is way larger than the cache size.

We see a slight increase (25%) in L1D cache. This degradation is compensated by reduced access time for lower level caches.

Average Miss Latency of L1D decrease while increases for L2C and LLC.

For bfs program, reducing cache size decreases the IPC. The reason is a slight increase (11%) in LLC misses and increase in average Miss Latency.

The reason for change in Average Miss latency is that the chances of finding the miss in next level cache reduces thus having to go further down.

IPC isn't affected by changing cache size in quicksort program.

We see a slight increase (18%) in L2C misses. This degradation is compensated by reduced access time for lower level caches.

Average Miss Latency increases. The reason is that the chances of finding the miss in next level cache reduces thus having to go further down.

2.4.4 Doubled Size

We see a slight decrease in MPKI (less than 1%), except in L2C of bfs program (significant decrease of 31%).

We also notice an increase in Average Miss Latency. The reason is due to increase in access time for larger caches.

Thus, we see no change in IPC except in bfs - decrease in MPKI results in improved IPC.

2.4.5 Reduced MSHR

IPC and MPKI remains constant to 4th decimal place. The reason is that decreasing MSHR size doesn't affect the number of misses in either of the programs - it might not get filled to even half of it.

There is minor decrease in average miss latency for reduced MSHR. This could be due to less time required to check the queue. But as the difference is less than a cycle, it doesn't affect the IPC.

2.4.6 Doubled MSHR

IPC and MPKI remains constant to 4th decimal place. The reason is that increasing MSHR size doesn't affect the number of misses in either of the programs - it might not get filled at all.

There is minor increase in average miss latency for doubled MSHR. This could be due to more time required to check the queue. But as the difference is less than a cycle, it doesn't affect the IPC.

*

Note: I haven't compared the L1I Average Miss Latency as the misses are single digit in most of the simulations which is insignificant compared to 10 million instructions

2.4.7 Plots

IPC

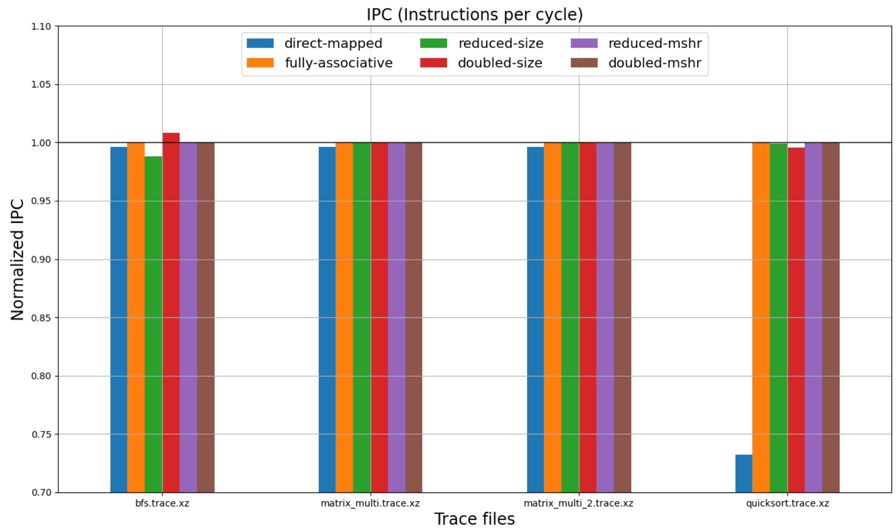

Figure 2.1: IPC (Instructions per cycle)

MPKI for L1 caches (L1D + L1I)

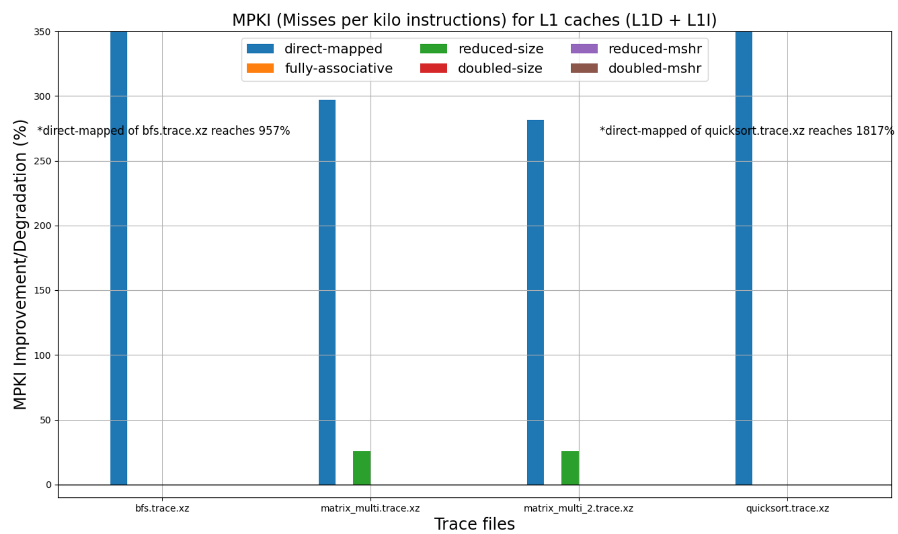

Figure 2.2: MPKI (Misses per kilo instructions) for L1 caches (L1D + L1I)

MPKI for L2 cache (L2C)

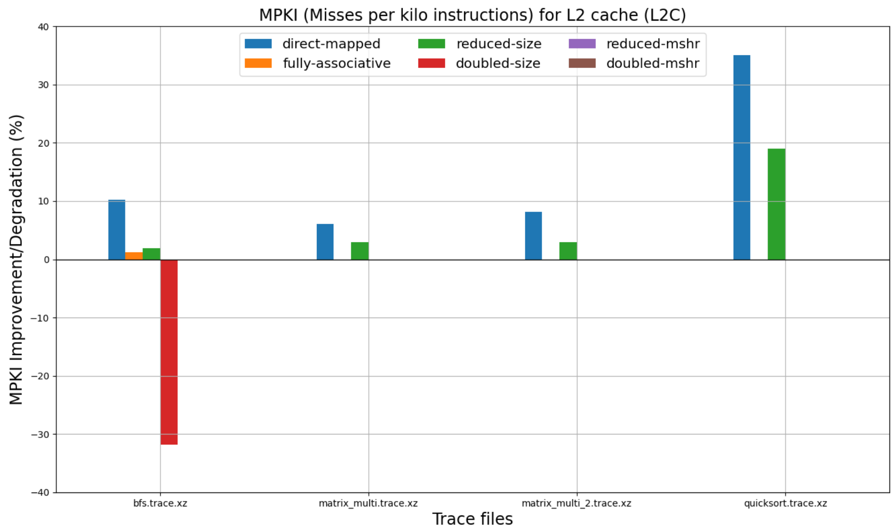

Figure 2.3: MPKI (Misses per kilo instructions) for L2 cache (L2C)

MPKI for L3 cache (LLC)

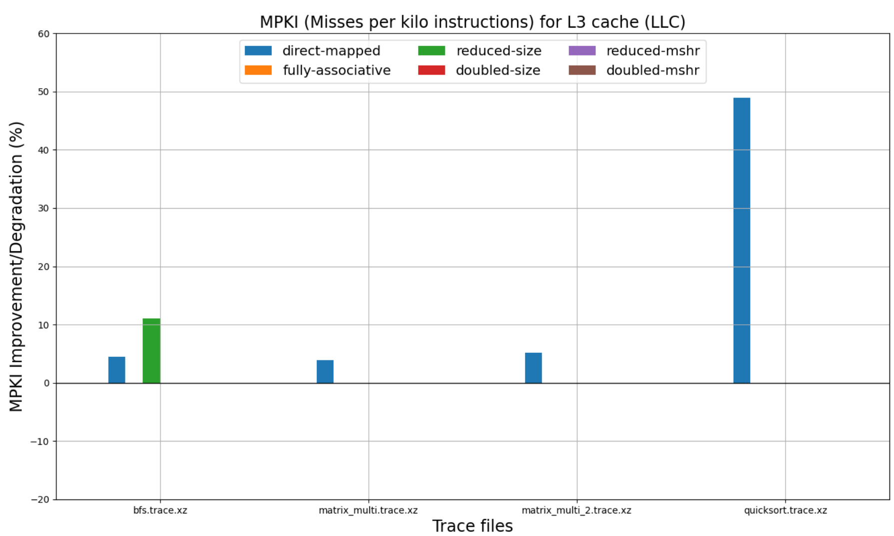

Figure 2.4: MPKI (Misses per kilo instructions) for L3 cache (LLC)