CS341: Computer Architecture Lab

Lab Assignment 4
Report

Aakriti
190050002

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2021-2022

Contents

Part 0: Getting Things Ready

1. Installed Intel VTune Profiler.
2. **How easy was it for you to install Vtune and get it up and running ? What were the challenges that you faced during the installation process and how much time did it take in total?**
   I prefer using windows as my PC is gets a bit heated up in Ubuntu so, I downloaded the Vtune profiler for Windows and just followed the basic instructions on the website. It requires VSCode to be installed which is quite a heavy application and takes a long time to download and even longer to get installed. Installing VSCode was more cumbersome than Vtune itself. After VSCode was installed, Vtune was a breeze. To get it up and running on my system, it took me a total of *6 hours*.
3. Installed Docker for operating system.

1. Part 1: Profiling with VTune

1.1 Performance Snapshot

1.1.1 bfs.cpp

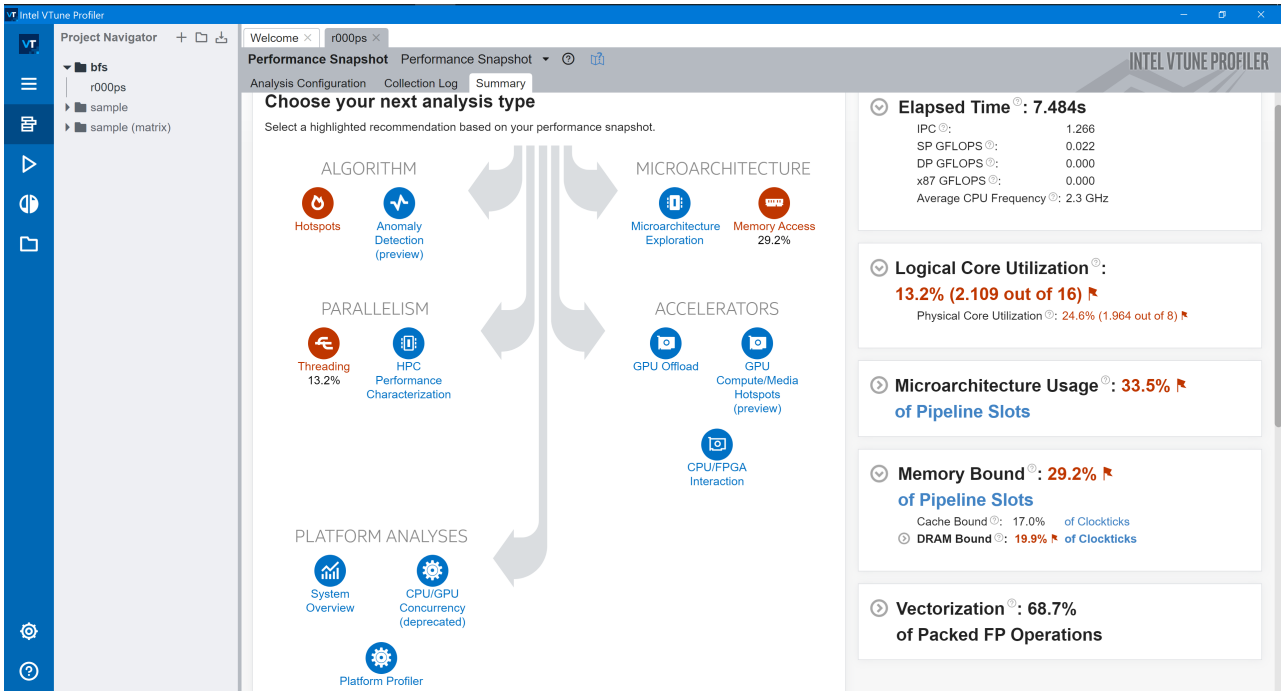

Figure 1.1: Performance Snapshot

1. IPC = 1.266
2. Logical core utilization = 13.2% (2.109 out of 16)
   Physical core utilization = 24.6% (1.964 out of 8)
3. %age of the Pipeline Slots that are Memory Bound = 29.2%
   Cache Bound = 17.0% of Clockticks
   DRAM Bound = 19.9% of Clockticks

1.1.2 quicksort.cpp

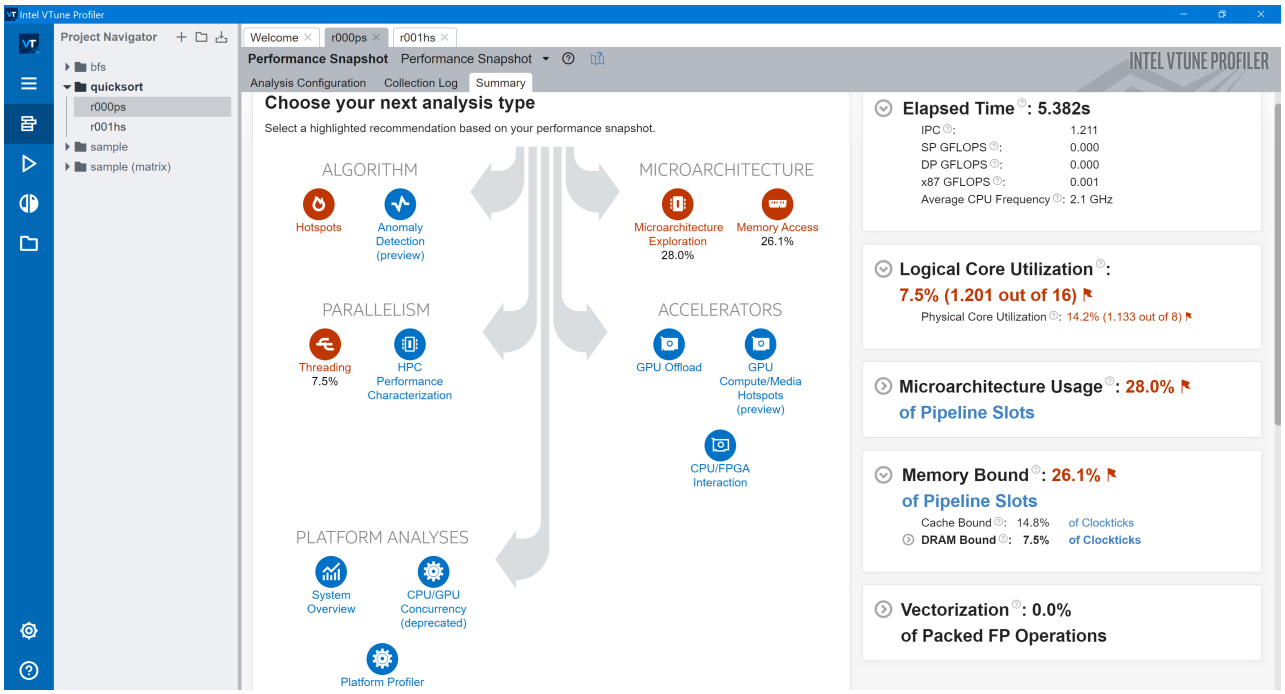

Figure 1.2: Performance Snapshot

1. IPC = 1.211

2. Logical core utilization = 7.5% (1.201 out of 16)
   Physical core utilization = 14.2% (1.133 out of 8)

3. %age of the Pipeline Slots that are Memory Bound = 26.1%
   Cache Bound = 14.8% of Clockticks
   DRAM Bound = 7.5% of Clockticks

1.1.3 matrix_multi.cpp

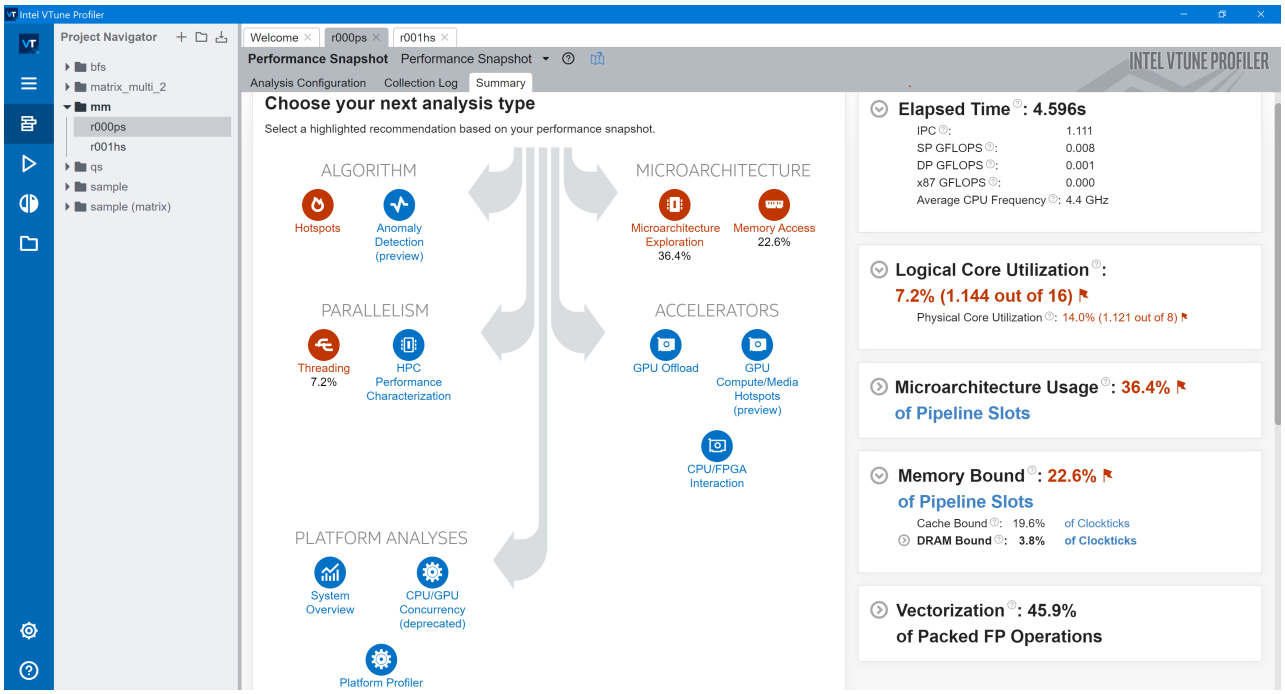

Figure 1.3: Performance Snapshot

1. IPC = 1.111

2. Logical core utilization = 7.2% (1.144 out of 16)
   Physical core utilization = 14.0% (1.121 out of 8)

3. %age of the Pipeline Slots that are Memory Bound = 22.6%
   Cache Bound = 19.6% of Clockticks
   DRAM Bound = 3.8% of Clockticks

1.1.4 matrix_multi_2.cpp

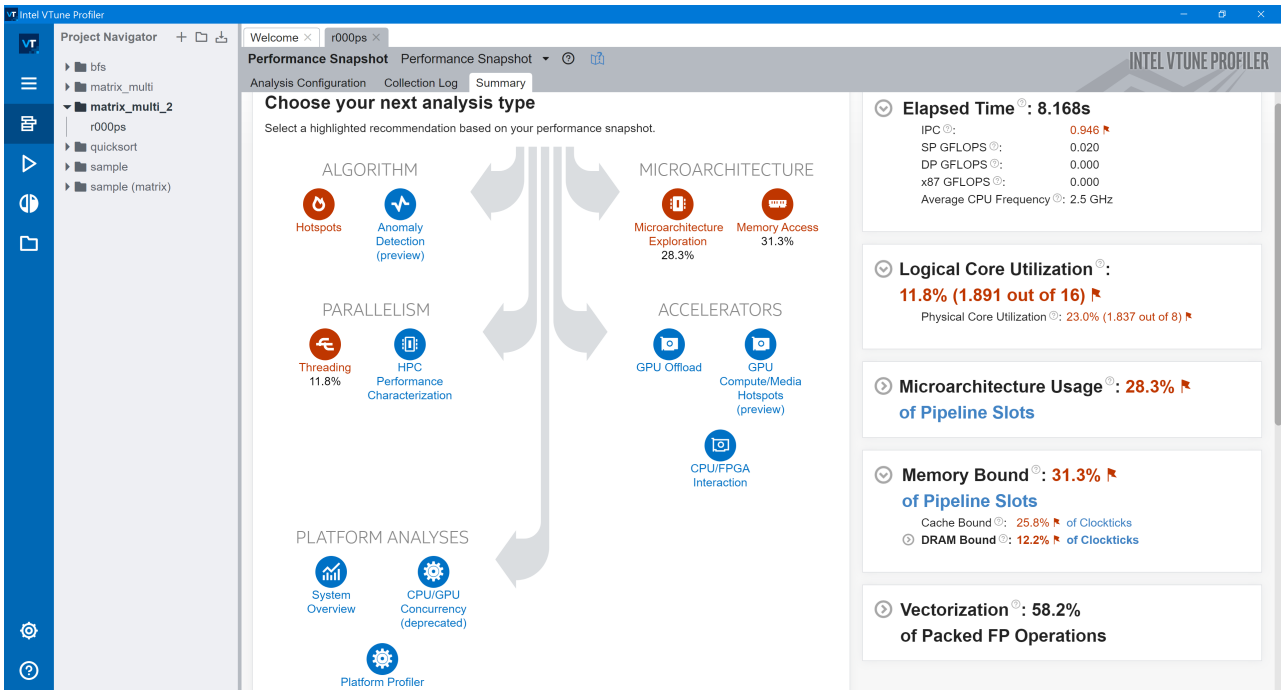

Figure 1.4: Performance Snapshot

1. IPC = 0.946
2. Logical core utilization = 11.8% (1.891 out of 16)
   Physical core utilization = 23.0% (1.837 out of 8)
3. %age of the Pipeline Slots that are Memory Bound = 31.3%
   Cache Bound = 25.8% of Clockticks
   DRAM Bound = 12.2% of Clockticks

1.2 Hotspots

1.2.1 bfs.cpp

1.2.1.1 Top hotspots along with their CPU time

Function	Module	CPU Time	%age CPU Time
bfs	bfs.exe	4.877s	48.7%
main	bfs.exe	2.588s	25.8%
malloc	msvcrt.dll	1.516s	15.1%
free	msvcrt.dll	0.659s	6.6%
__gnu_cxx::new_allocator<Node*>::construct<Node*, Node* const&>	bfs.exe	0.125s	1.2%

This is as expected because self calling recursive functions are expected to take most of the time if their recursion depth is high.

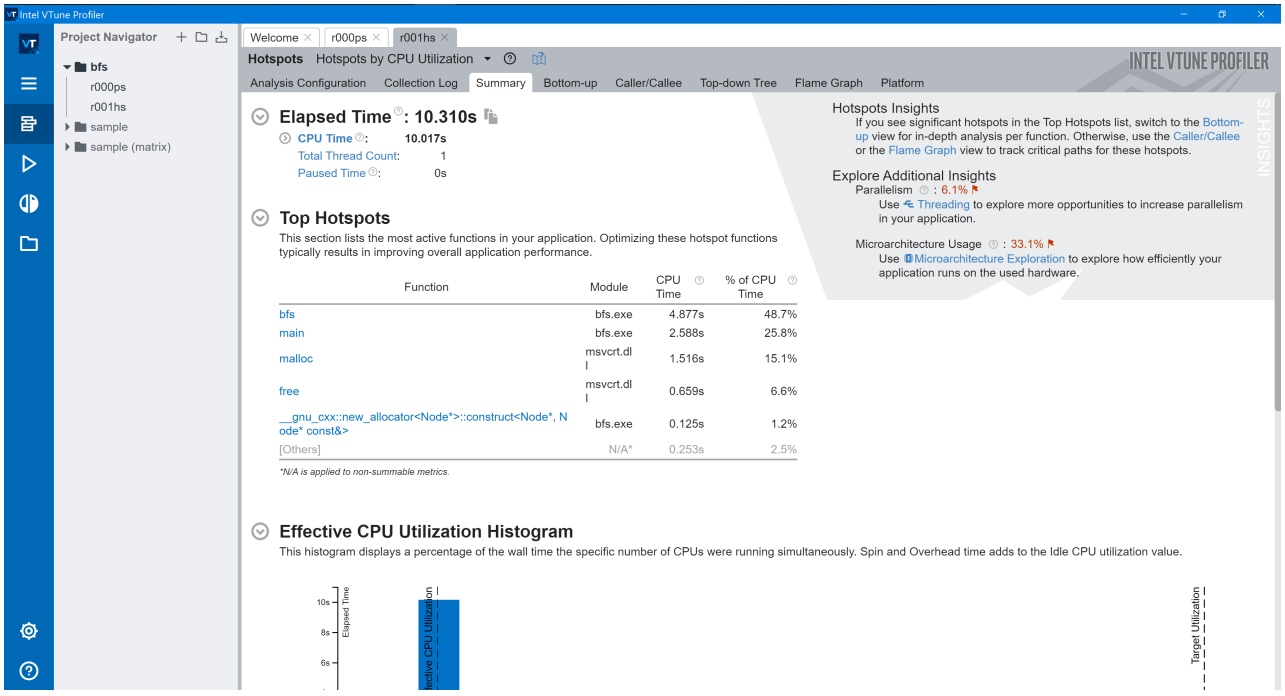

Figure 1.5: Hotspots

1.2.1.2 Statements responsible for consuming most of the CPU time

(This list ignores the statements that are from the source code of libraries)

Line	Source Code Line	CPU Time %	CPU Time (in s)
97	`right_child = curr_node->right;`	35.5%	3.558s
96	`left_child = curr_node->left;`	6.3%	0.627s
92	`for(int i=0; i<q_size; i++)`	3.9%	0.390s
99	`if(left_child) node_Q.push(left_child);`	1.4%	0.140s
100	`if(right_child) node_Q.push(right_child);`	1.2%	0.116s
93	`curr_node = node_Q.front();`	0.5%	0.046s

Because there are not a lot of arithmetic and time consuming operations in this code, we expect memory access to take the largest amount of time leading to the high CPU time for accessing the right child.

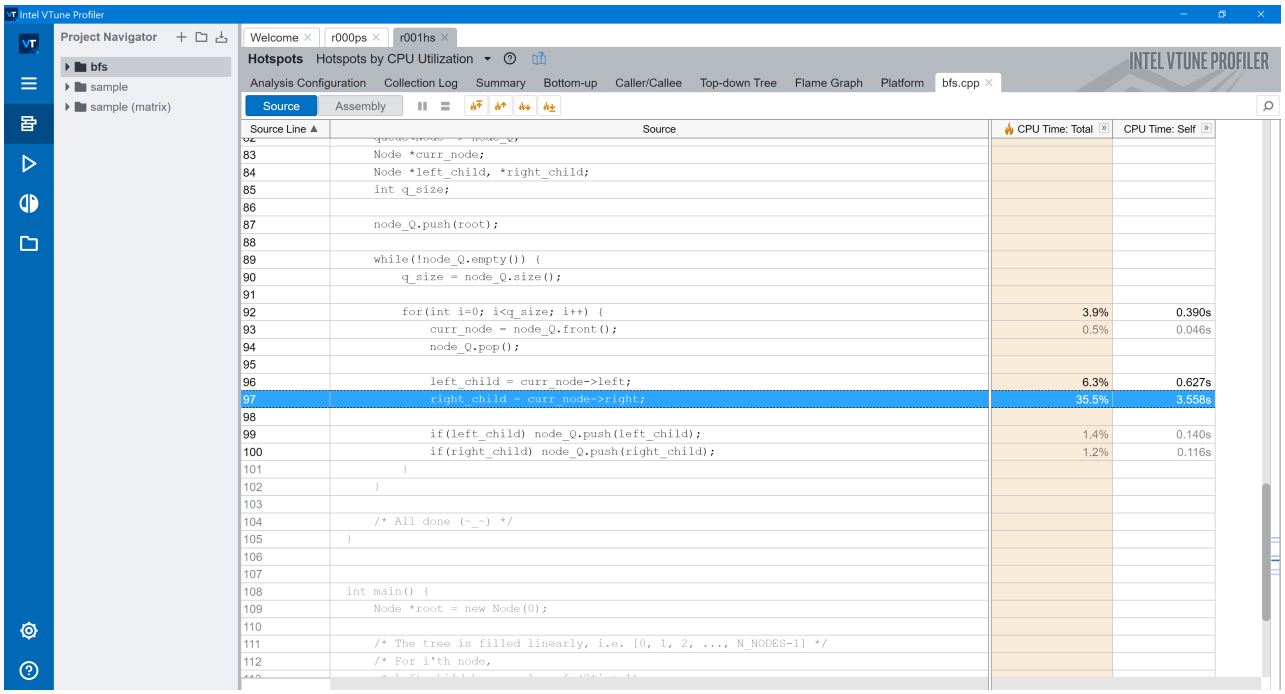

Figure 1.6: Source code and CPU time consumed

1.2.2 quicksort.cpp

1.2.2.1 Top hotspots along with their CPU time

Function	Module	CPU Time	%age CPU Time
quicksort	quicksort.exe	1.663s	62.7%
partition	quicksort.exe	0.703s	26.5%
swap	quicksort.exe	0.160s	6.0%
free	msvcrt.dll	0.105s	4.0%
read_encoded_value_with_base	libgcc_s_dw2-1.dll	0.015s	0.6%

Quicksort is a recursive function that calls itself multiple times, and this code has no additional significant processing. As a result, quicksort is predicted to consume more CPU time, which it does.

`partition` and `swap` are being called at aevery recursive call of the quicksort function. Since, both of these functions take arguments by value and not by reference, it is expected to take a lot of time.

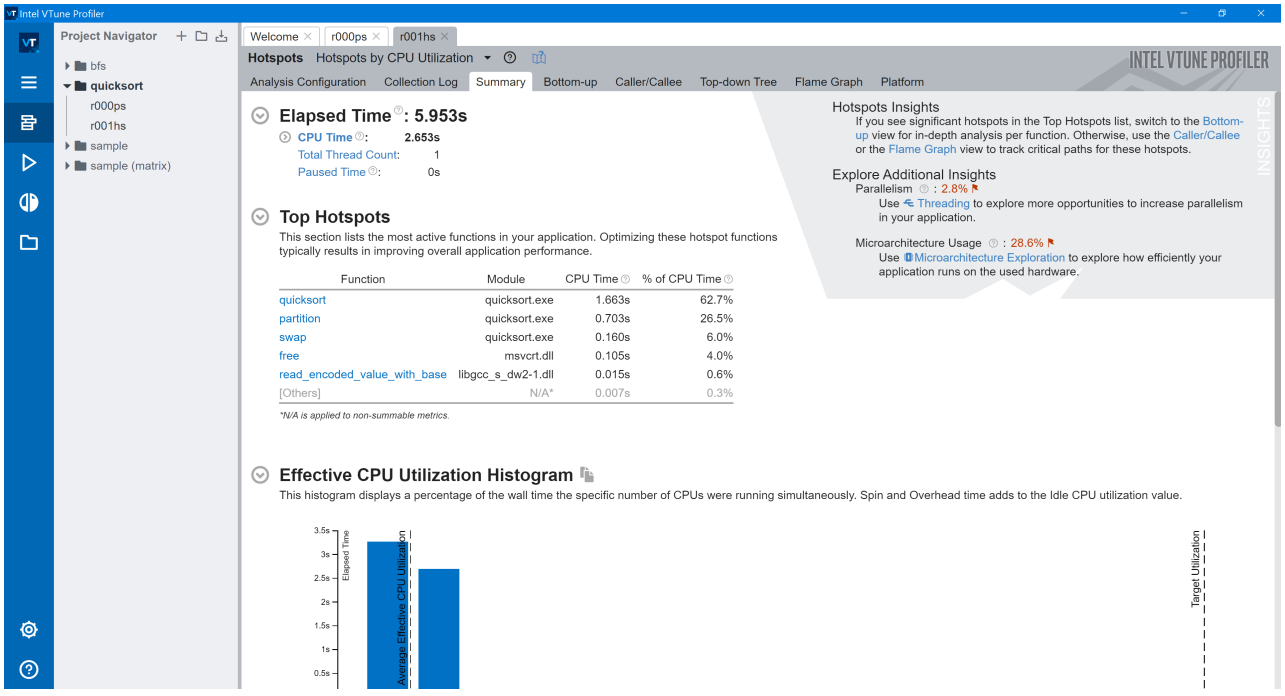

Figure 1.7: Hotspots

1.2.2.2 Statements responsible for consuming most of the CPU time

(This list ignores the statements that are from the source code of libraries)

Line	Source Code Line	CPU Time %	CPU Time (in s)
45	`quicksort(nums, lo, p-1);`	62.7%	1.165s
44	`long p = partition(nums, lo, hi);`	18.8%	0.498s

As explained above, the same reasoning takes a lot of time here.

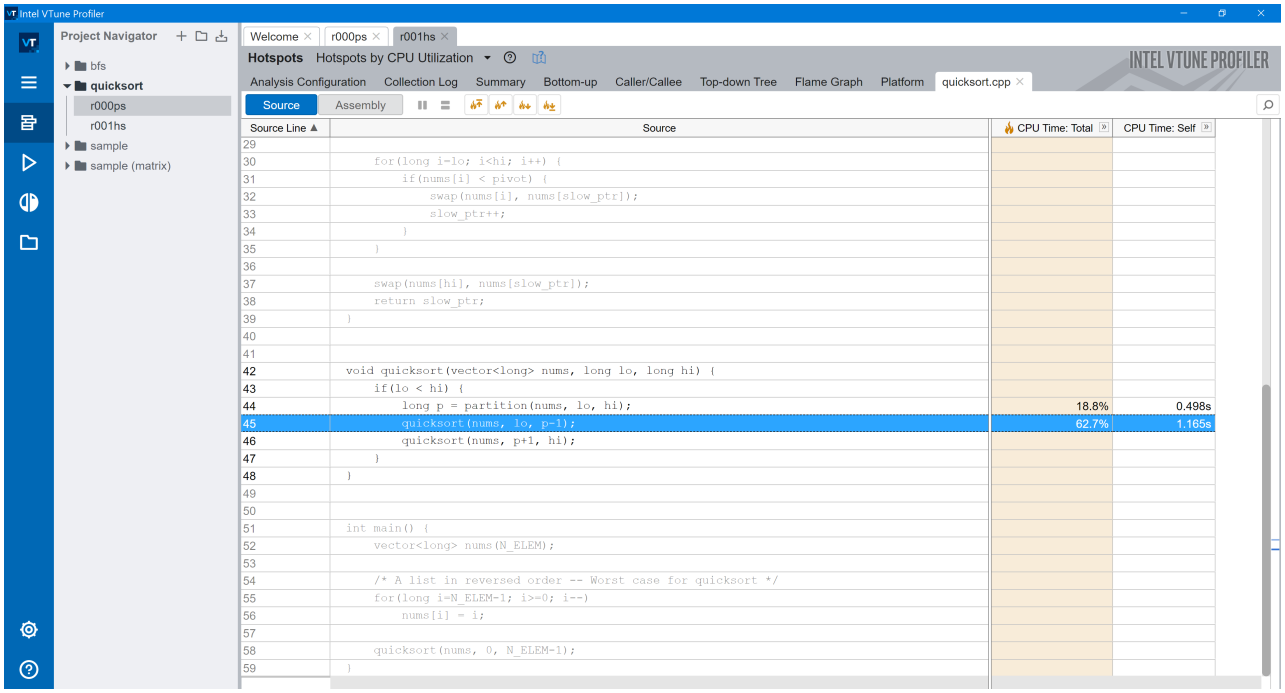

Figure 1.8: Source code and CPU time consumed

1.2.3 matrix_multi.cpp

1.2.3.1 Top hotspots along with their CPU time

Function	Module	CPU Time	%age CPU Time
matrix_product	matrix_multi.exe	4.325s	100.0%

Since most of the work is done inside the matrix product function, it is obvious that it will occupy almost all of the CPU time.

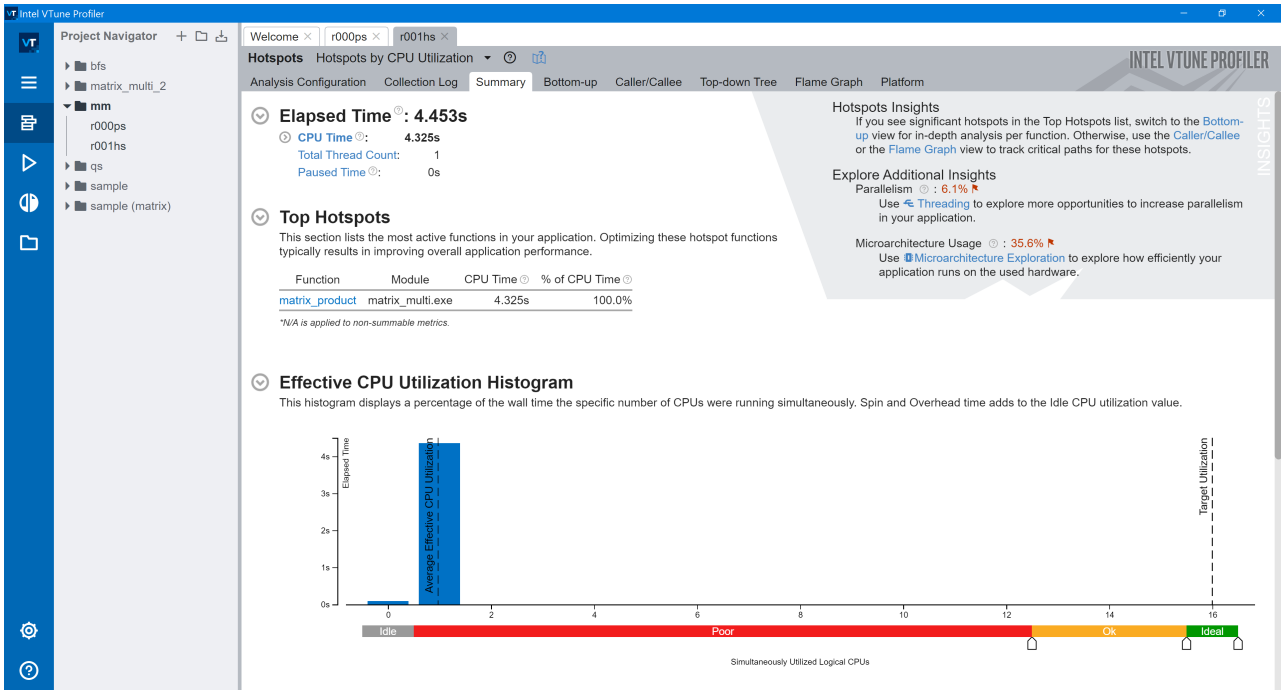

Figure 1.9: Hotspots

1.2.3.2 Statements responsible for consuming most of the CPU time

(This list ignores the statements that are from the source code of libraries)

Line	Source Code Line	CPU Time %	CPU Time (in s)
32	`C[i][j] += A[i][k] * B[k][j];`	92.0%	3.981s
31	`for(int k=0; k<N_DIMS; k++)`	8.0%	0.344s

While this code contains for loops, the variables are local, so we don't anticipate to spend a lot of time incrementing and verifying whether the for loop's condition is true or not. We anticipate that accessing the matrix's elements will take longer.

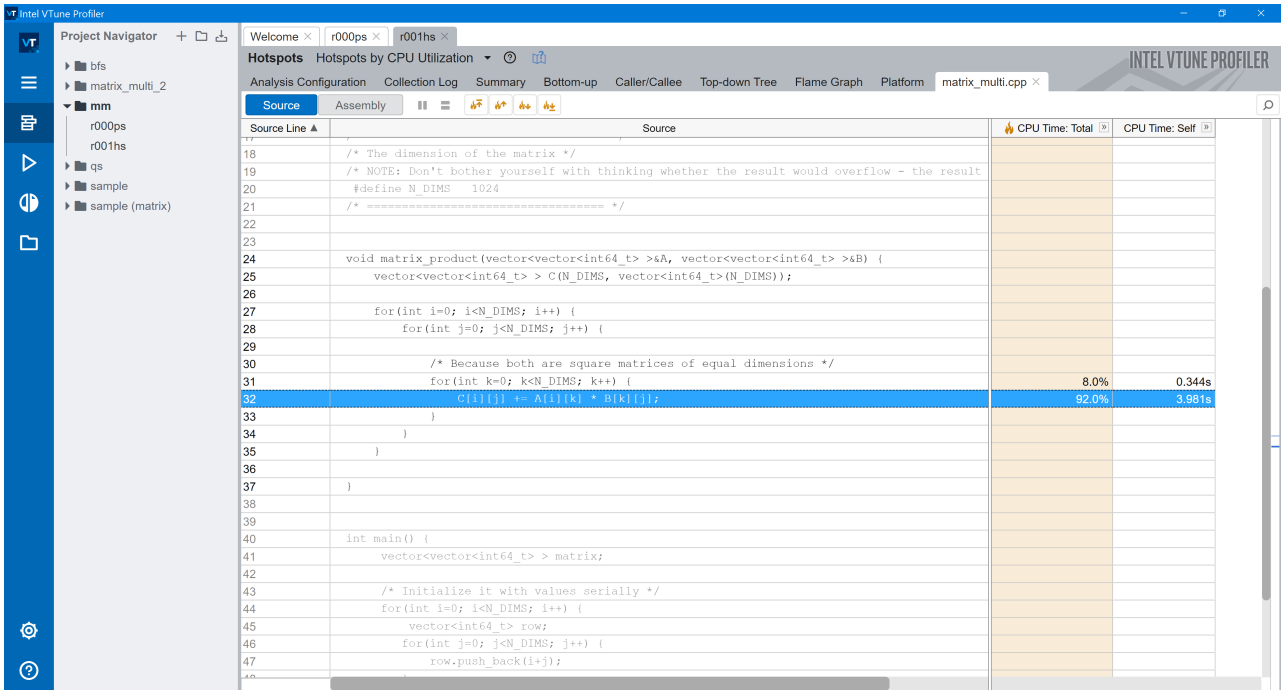

Figure 1.10: Source code and CPU time consumed

1.2.4 matrix_multi_2.cpp

1.2.4.1 Top hotspots along with their CPU time

Function	CPU Time	%age CPU Time
matrix_product	14.080s	98.9%
std::vector<long long, std::allocator<long long>>::operator[]	0.151s	1.1%

Since most of the work is done inside the matrix product function, it is obvious that it will occupy almost all of the CPU time.

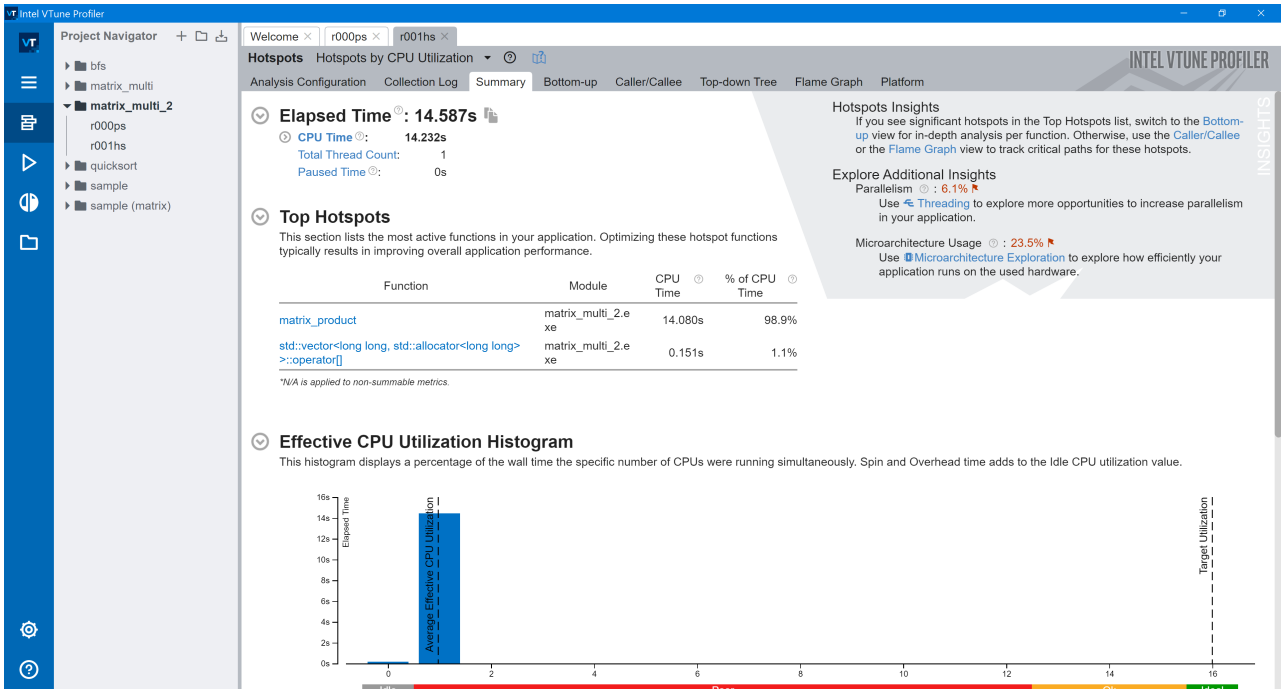

Figure 1.11: Hotspots

1.2.4.2 Statements responsible for consuming most of the CPU time

(This list ignores the statements that are from the source code of libraries)

Line	Source Code Line	CPU Time %	CPU Time (in s)
32	`C[i][j] += A[i][k] * B[k][j];`	92.7%	13.196s
31	`for(int k=0; k<N_DIMS; k++)`	6.2%	0.884s

While this code contains for loops, the variables are local, so we don't anticipate to spend a lot of time incrementing and verifying whether the for loop's condition is true or not. We anticipate that accessing the matrix's elements will take longer.

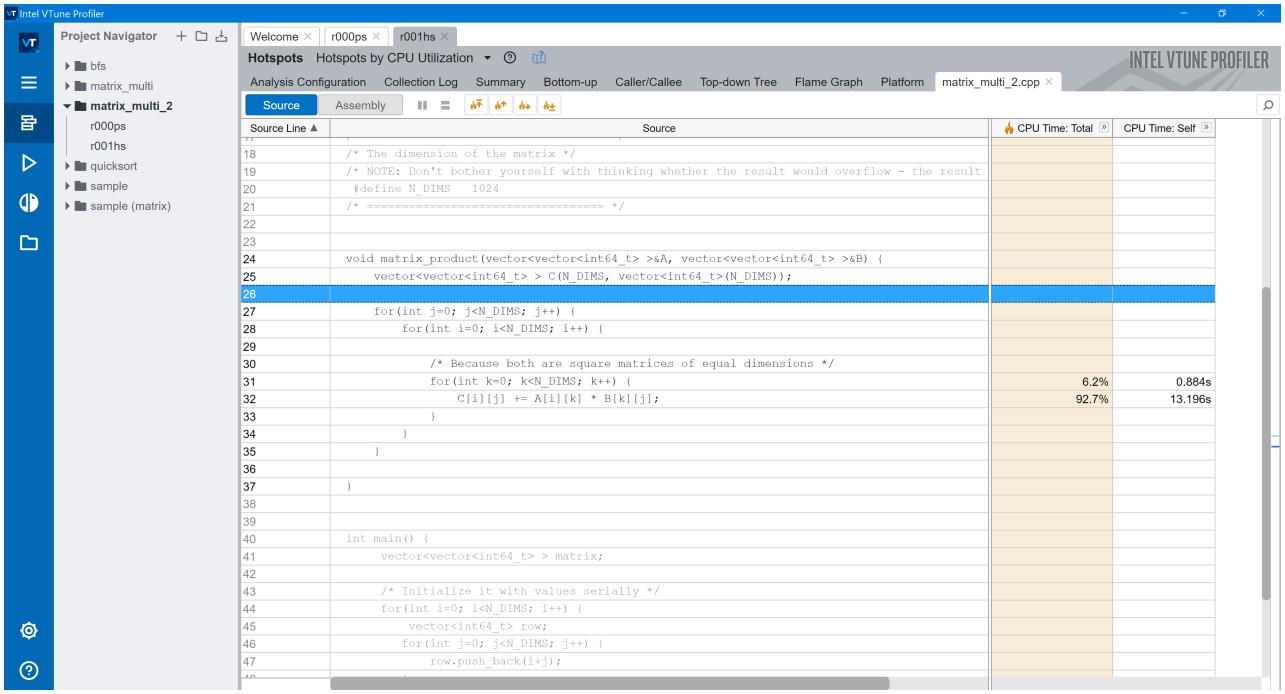

Figure 1.12: Source code and CPU time consumed

2. Simulating with ChampSim

2.1 Experiments

The specific configurations -

- Baseline : No change
- Direct mapping : For each cache,
  sets_new = sets_old × ways_old
  ways_new = 1
- Fully associated : For each cache,
  ways_new = sets_old × ways_old
  set_new = 1
- Double MSHR and Halve MSHR : Directly double or halve MSHR values as required.
- Double sets and halve sets : Calculate latency using CACTI and round to an integer.

Cache	Default Latency	Access Time (in ns)			New Latency (in Cycles)	
	(in cycles)	Default	Half	Double	Half	Double
L1I	4	2.29369	2.18023	2.42581	[3.80] = 4	[4.230] = 4
L1D (8 associativity)	5	2.29369	2.18023	2.42581		
L1D (16 associativity)	5	4.44389	4.33016	4.5762		
L1D (average of 8 and 16)	5	3.36879	3.25519	3.50101	[4.561] = 5	[5.1961] = 5
L2	10	3.13704	2.77869	3.95873	[8.858] = 9	[12.619] = 13
L3 (LLC)	20	6.19609	5.4195	7.3387	[17.493] = 17	[23.688] = 24

Figure 2.1: CACTI results

Steps followed

1. Run ./make tracer.sh (in Champsim/tracer)
2. Make traces of all programs by using
   `pin -t obj-intel64/champsim_tracer.so -o ../../traces/quicksort.trace -t 20 -- quicksort.out`

3. For each specific configuration required as mentioned above -

    - Modify the cache.h file as required
    - Run `./build champsim.sh bimodal no no no no lru 1`
    - Run `./run champsim.sh bimodal-no-no-no-no-lru-1core 10 10 program.trace.xz` for each program bfs, both matrix multiplications and quicksort
    - Copy the results from result 10M to the required directory

2.2 Results

Here MKPI is negative as we see an improvement so there should be less misses per kilo instructions

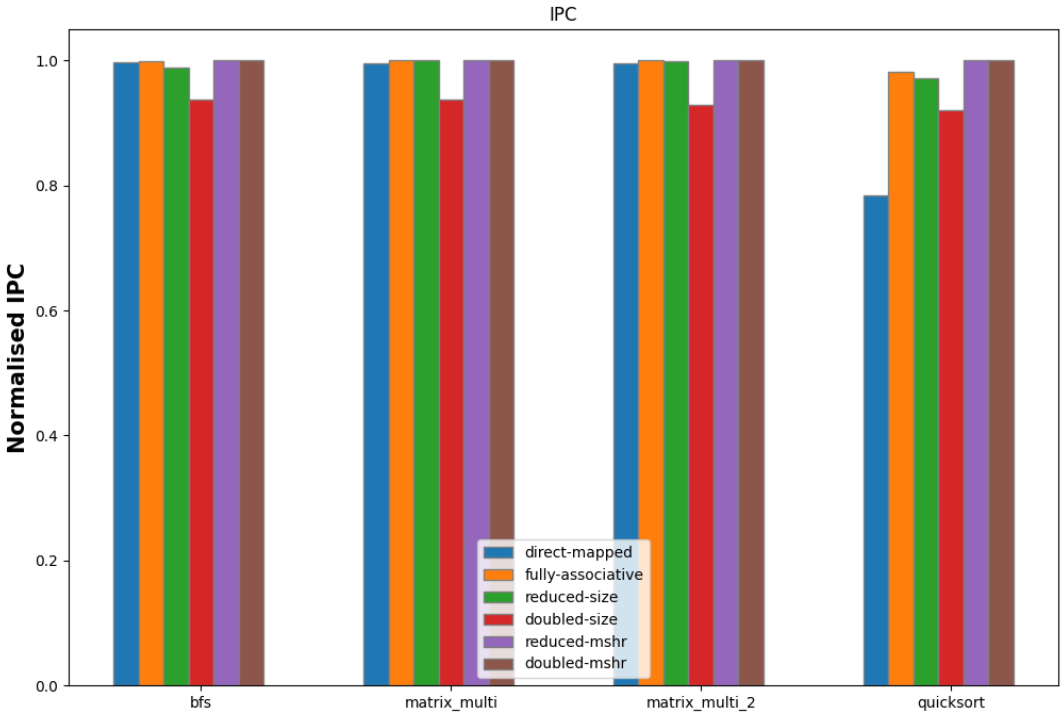

Figure 2.2: Normalized IPC

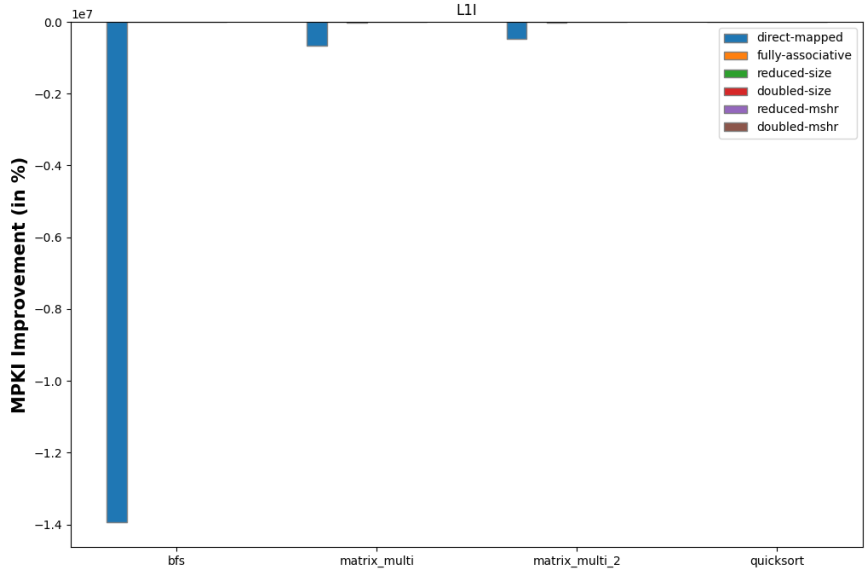

Figure 2.3: MKPI Improvement - L1I

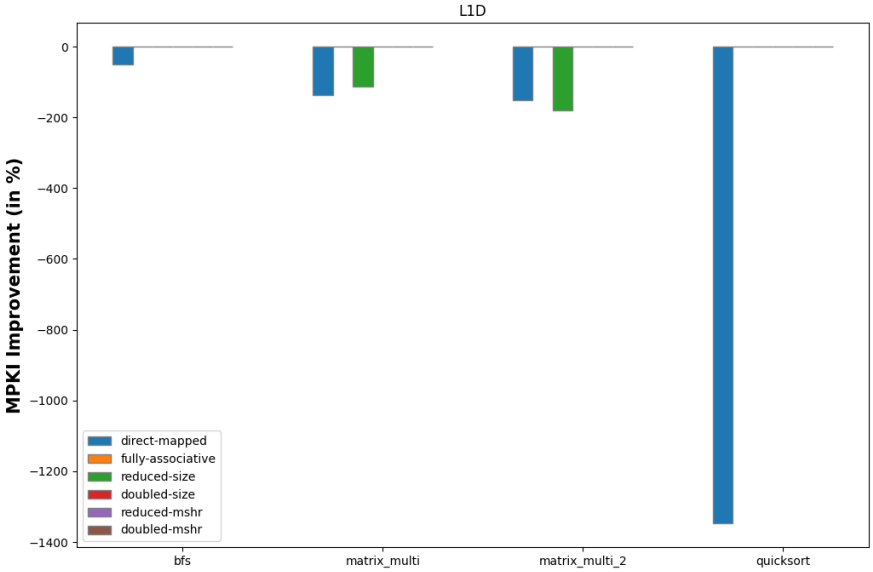

Figure 2.4: MKPI Improvement - L1D

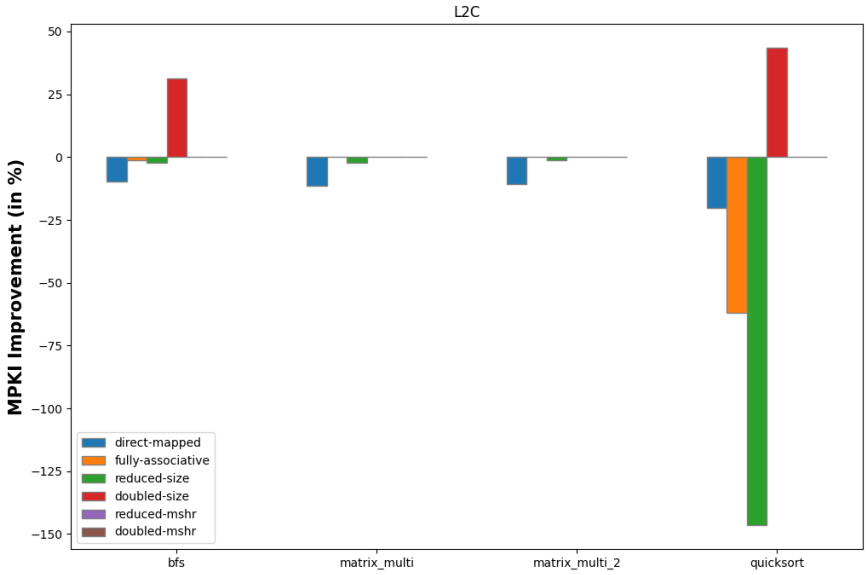

Figure 2.5: MKPI Improvement - L2

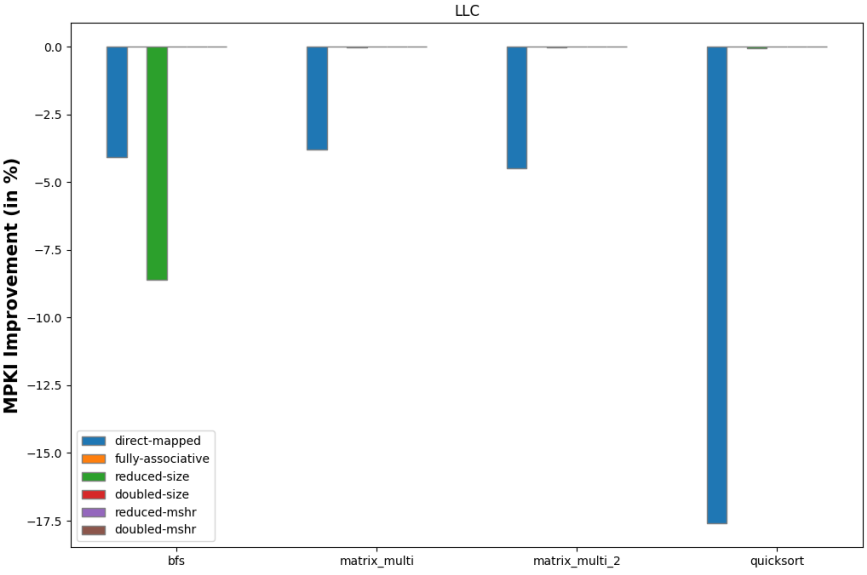

Figure 2.6: MKPI Improvement - LLC

2.3 Observations

2.3.1 Direct Mapping

For bfs, matrix-multi, and matrix-multi-2, IPC fell somewhat. Quicksort, on the other hand, has an IPC ratio of around 0.8. A memory region is not visited very frequently in all three of bfs and both matrix multiplications. Both associative caches and direct mapping flush it in the middle. However, because of its direct mapping nature, direct mapping removes high-reuse material in quicksort, where memory regions are often reused. As a result, more time is spent requesting memory, resulting in a lower IPC indirectly.

When compared to the baseline, the average miss latency has lowered across all caches. MKPIs values are much much lower than before in all programs. This is because the comparators do not need to spend any more time comparing whether the tag is equal to the required tag. As a result, the hit time lowers across all levels, and the process of pinging the lower layer cache if there is a miss from L1 to the DRAM takes less time.

2.3.2 Fully Associative Cache

The IPC has more or less remained the same as the baseline. The MPKIs have also remained more or less the same as the baseline for all except L2.

This improvment is quite significant in case of quicksort, although bfs also shows slight improvement. Since, in most of the cases MKPI has remained the same, observations for fully associative seem inconclusive. However, we can say that if memory regions are reused as in quicksort, L2 benefits from full associativity.

2.3.3 Reduced Size

There has been a slight increase in IPC for each program by about 5-8%.

Average latency decreases for all except L1I because MKPI decreased for all except L1I.

Average MKPI for all programs across all caches shows an improvement that is reduction, as these are small programs memory access coded in such a manner that it benefits from smaller size and frequent replacements.

The MPKI has remained about the same for bfs and quicksort but decreased by about 70% for both matrix multiplications for L1D. It improves MKPI for L2 in quicksort and LLC in bfs by a very good fraction.

Both the matrix multiplication codes benefit across all caches, in some fractions because they benefit from frequent replacement.

2.3.4 Doubled Size

The IPC has decreased by about 5-8% for bfs, matrix multiplications and for quicksort.

The MPKI for L1I are approximately 0 for both the baseline and the doubled size.

MPKI has increased for L2. This is easily explained by the fact that more sets amounts to greater retention of high use values.
Because the sample size of L1I and L1D misses are too low, it is difficult to observe anything significant about them. For the other caches however like L2, the average miss latency has increased which was expected due to the higher hit time required.

2.3.5 Reduced MSHR

The IPC for the decreased MSHR is within 1 percent of the baseline IPC.
All MPKIs are essentially the same. We expect the hit and miss rates to be the same because MSHR has no effect on associativity, set size, or the amount of entries in the cache. And it lives up to our expectations. In addition, the average miss delay is the same. We anticipated the average miss delay to remain the same because MSHRs have no effect on hit time or miss time.

2.3.6 Doubled MSHR

We do not expect changes in the number of MSHRs to alter the miss rate, hit rate, hit time, or miss time, as we did with the decreased MSHR. As a result of the similarity of all the variables, IPC, MPKI, and average miss cycle are all within 1 percent of the baseline values.