CS341: Computer Architecture Lab

Lab 4: VTune Profiling and ChampSim
Report

Adarsh Raj (190050004)

October 2021

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

2021-2022

Contents

1. Experiments

1.1 Profiling With VTune

For this task, the default values of cpp programs were changed such that the execution time, took 3-10 seconds **with** compiler optimization and debugging flags (i,e -g and -O2 flags). There values were changed as follows:

1. `bfs.cpp` - No change.
2. `quicksort.cpp` - Number of elements changed to $(2^{14} + 2^{11})$, from 2^{20}.
3. `matrix_multi.cpp` - Dimension Size changed to 1200 from 1024.
4. `matrix_multi_2.cpp` - Dimension Size changed to 1200 from 1024.

The results of Performance Analysis and Hotspots are provided in the Results section, along with the screenshots.

1.2 Simulating with ChampSim

For this task, the default values of cpp programs were changed such that the execution time, took 3-10 seconds **without** compiler optimization and debugging flags (i,e -g and -O2 flags). There values were changed as follows:

1. `bfs.cpp` - Number of elements changed to 2^{14} from 2^{20}, and number of bfs iterations changed to 800 from 1000.
2. `quicksort.cpp` - Number of elements changed to $(2^{13} + 2^{12})$, from 2^{20}.
3. `matrix_multi.cpp` - Dimension Size changed to 500 from 1024.
4. `matrix_multi_2.cpp` - Dimension Size changed to 500 from 1024.

The results of simulation from ChampSim for different cases, i.e baseline, direct-mapped, fully-associative, etc are in the Results section, along with plots to express changes across different cases.

2. Getting Things Ready

Installation Of Vtune

Installed Vtune for Windows from components page of Intel OneApi base toolkit.

Challenges in Installation Phase

During the installation I faced the following challenges:

1. At first, I tried to install Vtunes on my Ubuntu dual boot system, but after installing through script, there were errors while configuring the vars for vtune gui.
2. After trying for a few hours, finally it worked and i opened GUI only to realise that my ubuntu system does not allow Performance analysis directly. It aksed me to setup kernel/yama and paranoid flag of ubuntu system which I was not having any idea about.
3. For one or two hours I tried to change Ubuntu system variables to make VTune work, but it does not resulted in my favour.
4. So I decided to ditch Ubuntu, and try installation on Windows, which worked quite well. Although after this, I had to install MINGW compiler for c/c++ support in my Windows.

Docker and ChampSim Installation

Installed Docker from the provided link and pulled the image `0xd3ba/champsim-lab`.

3. Results

3.1 Profiling With VTune - Performance Analysis

- `bfs.cpp`: The screenshot for the performance analysis for this program with optimization flags is as follows:

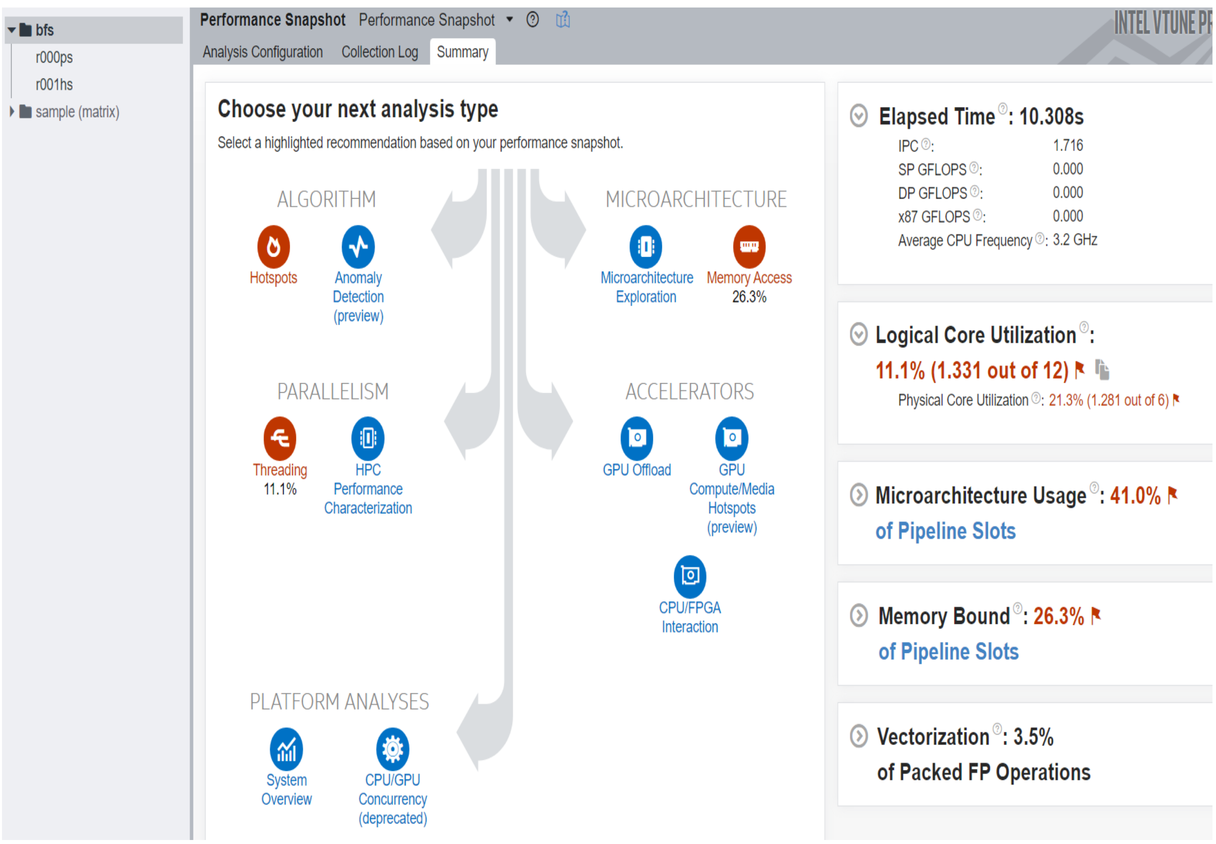


Figure 3.1: Performance Analysis of BFS Algorithm

1. The elapsed time for the program was **10.308 secs** and the IPC observed was **1.716**.
2. Logical Core Utilization - **11.1 percent** (1.331 out of 12).
3. Physical Core Utilization - **21.3 percent** (1.281 out of 6).
4. The percentage of memory bound pipeline slots are **26.3** percent.

- `matrix_multi.cpp`: The screenshot for the performance analysis for this program with optimization flags and change in number of dimensions from 1024 to 1200, is as follows:

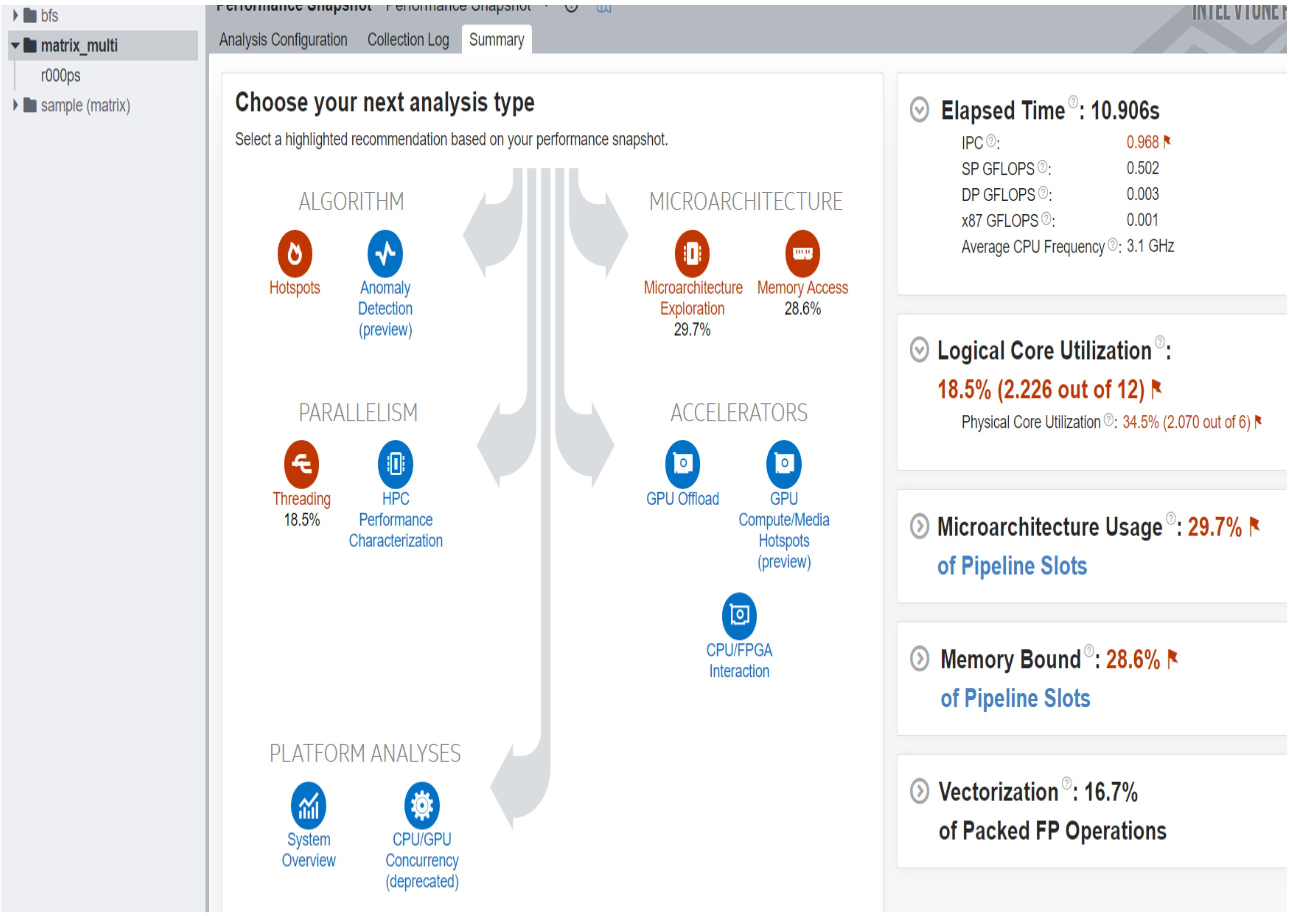


Figure 3.2: Performance Analysis of 1st Matrix Multi Algorithm

1. The elapsed time for the program was **10.906 secs** and the IPC observed was **0.968**.
2. Logical Core Utilization - **18.5 percent** (2.226 out of 12).
3. Physical Core Utilization - **34.5 percent** (2.070 out of 6).
4. The percentage of memory bound pipeline slots are **28.6** percent.

- `matrix_multi_2.cpp`: The screenshot for the performance analysis for this program with optimization flags and change in number of dimensions from 1024 to 1200, is as follows:

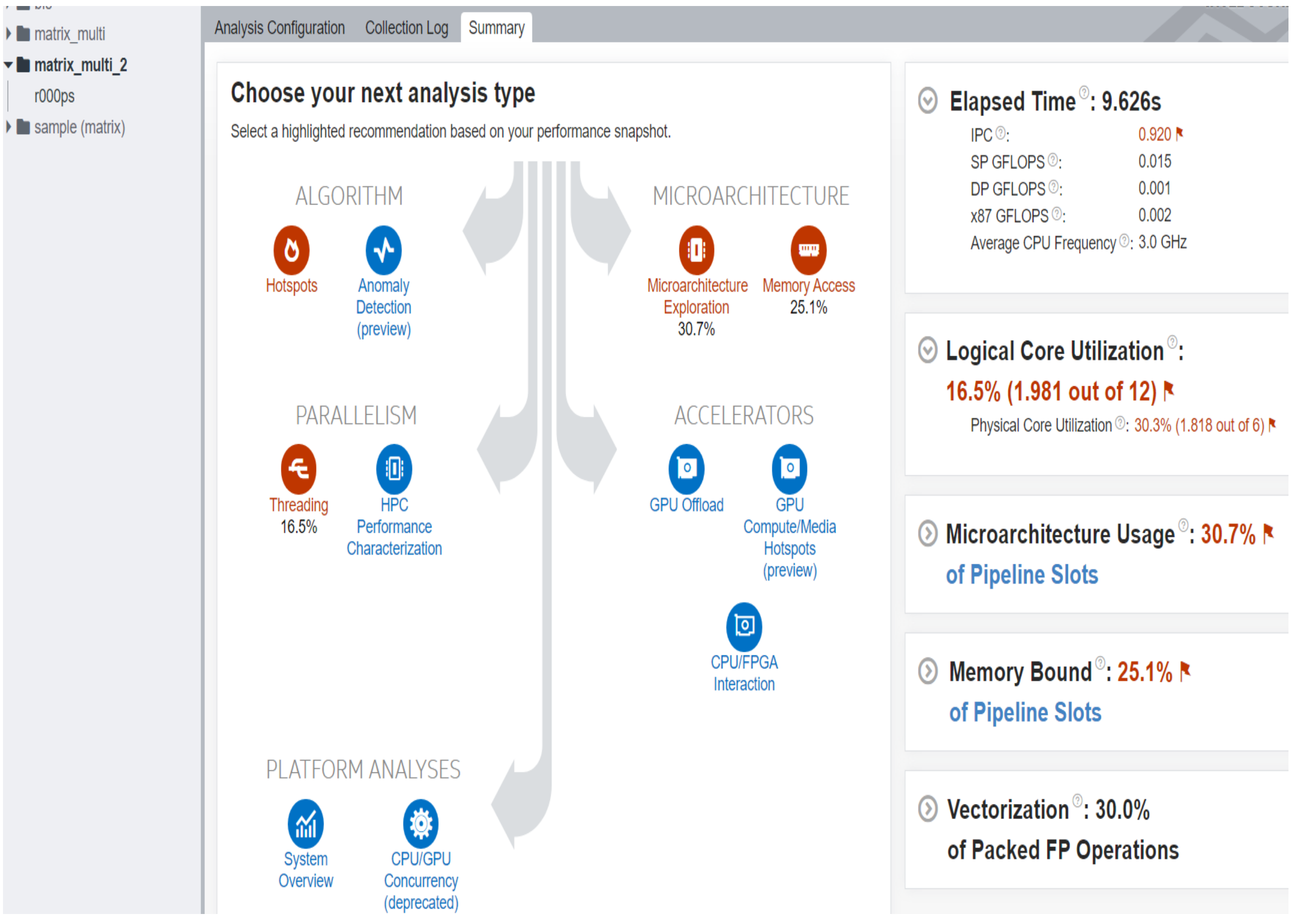


Figure 3.3: Performance Analysis of 2nd Matrix Multi Algorithm

1. The elapsed time for the program was **9.626 secs** and the IPC observed was **0.920**.
2. Logical Core Utilization - **16.5 percent** (1.981 out of 12).
3. Physical Core Utilization - **30.3 percent** (1.818 out of 6).
4. The percentage of memory bound pipeline slots are **25.1** percent.

- `quicksort.cpp`: The screenshot for the performance analysis for this program with optimization flags and change in number of elements from 2^{20} to $(2^{14} + 2^{11})$, is as follows. Note that, due to large number of elements, I was getting `bad_alloc()` error in Windows which got resolved only on decreasing the number of elements to the above value. Increasing the value by two times or more than the set value also resulted in the same error.

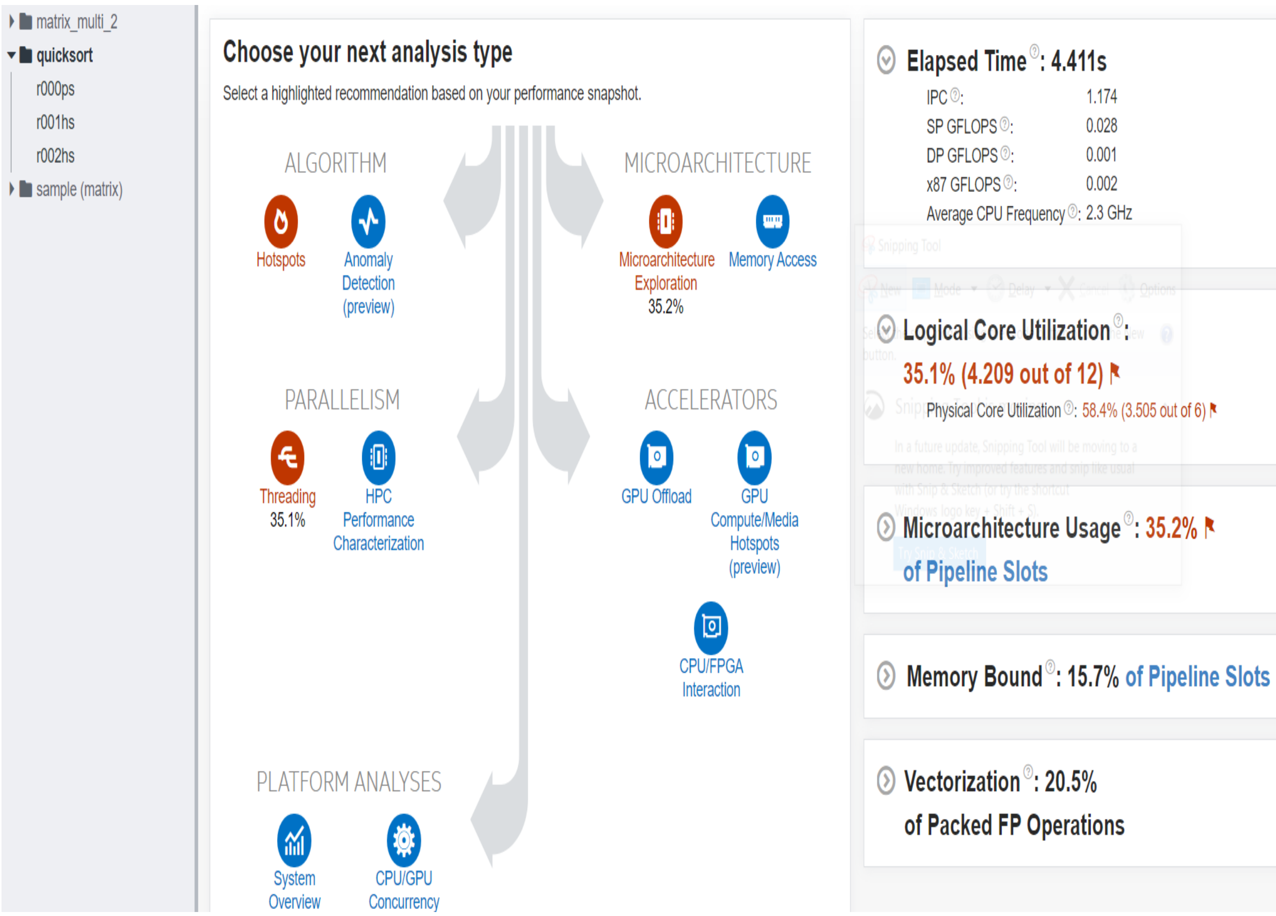


Figure 3.4: Performance Analysis of QuickSort Algorithm

1. The elapsed time for the program was **4.411 secs** and the IPC observed was **1.174**.
2. Logical Core Utilization - **35.1 percent** (4.209 out of 12).
3. Physical Core Utilization - **58.4 percent** (3.505 out of 6).
4. The percentage of memory bound pipeline slots are **15.7** percent.

3.2 Profiling With VTune - Hotspots

- **BFS Algorithm** (`bfs.cpp`) The top hotspots that were identified along with their CPU time can be found in the screenshot of hotspot test for the program as follows:

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
bfs	bfs.exe	6.039s	52.0%
main	bfs.exe	2.567s	22.1%
malloc	msvcrt.dll	1.827s	15.7%
free	msvcrt.dll	0.694s	6.0%
__gnu_cxx::new_allocator<Node*>::construct<Node*, Node* const&>	bfs.exe	0.172s	1.5%
[Others]	N/A*	0.326s	2.8%

*N/A is applied to non-summable metrics.

Figure 3.5: Hotspots for BFS Algorithm

The statements in the program's source code that were responsible for consuming most of the CPU time (in descending order of the % of CPU time consumed) are as follows:

Line	Code	CPU Time (Total)	CPU Time (Self)
97	`right_child = curr_node->right;`	39.1 %	4.543 s
120	`bfs(root);`	22.1 %	2.567 s
96	`left_child = curr_node->left;`	5.8 %	0.680 s
92	`for(int i=0;i<q_size;i++) {`	4.5 %	0.528 s
100	`if(right_child) node_Q.push_back(right_child);`	1.1 %	0.123 s
99	`if(left_child) node_Q.push_back(left_child);`	1.0 %	0.119 s
93	`curr_node = node_Q.front();`	0.4 %	0.046 s

Screenshots supporting the above table are attached as follows:

Source Line	Source	CPU Time: Total	CPU Time: Self
97	right_child = curr_node->right;	39.1%	4.543s
96	left_child = curr_node->left;	5.8%	0.680s
92	(int i=0; i<q_size; i++) {	4.5%	0.528s
100	if(right_child) node_Q.push(right_child);	1.1%	0.123s
99	if(left_child) node_Q.push(left_child);	1.0%	0.119s
93	curr_node = node_Q.front();	0.4%	0.046s
7	kes kind of 10+ seconds, you're good to go.		

Figure 3.6: bfs function - Hotspot

Source Line	Source	CPU Time: Total	CPU Time: Self
120	bfs(root);	22.1%	2.567s
2	* bfs.cpp -- Simple pro		

Figure 3.7: main function - Hotspot

- **Matrix Multiplication-1** (`matrix_multi.cpp`) The top hotspots that were identified along with their CPU time can be found in the screenshot of hotspot test for the program as follows:

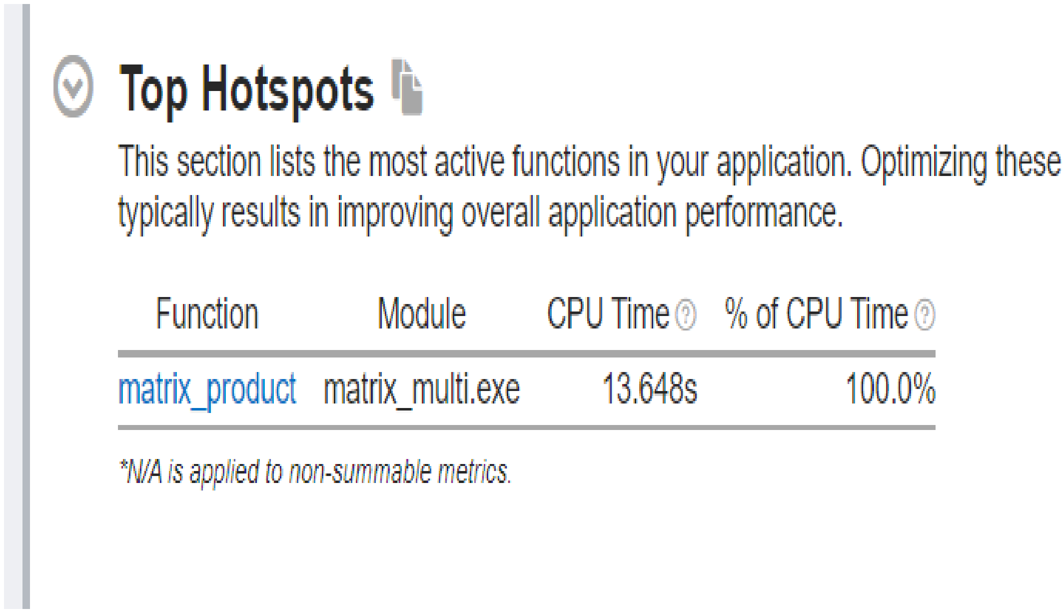


Figure 3.8: Hotspots for 1st Matrix Multiplication Algorithm

The statements in the program's source code that were responsible for consuming most of the CPU time (in descending order of the % of CPU time consumed) are as follows:

Line	Code	CPU Time (Total)	CPU Time (Self)
32	`C[i][j] += A[i][k]*B[k][j]`	94.2 %	12.862 s
31	`for(int k=0; k<N_DIMS; k++) {`	5.8 %	0.786 s

Screenshots supporting the above table are attached as follows:

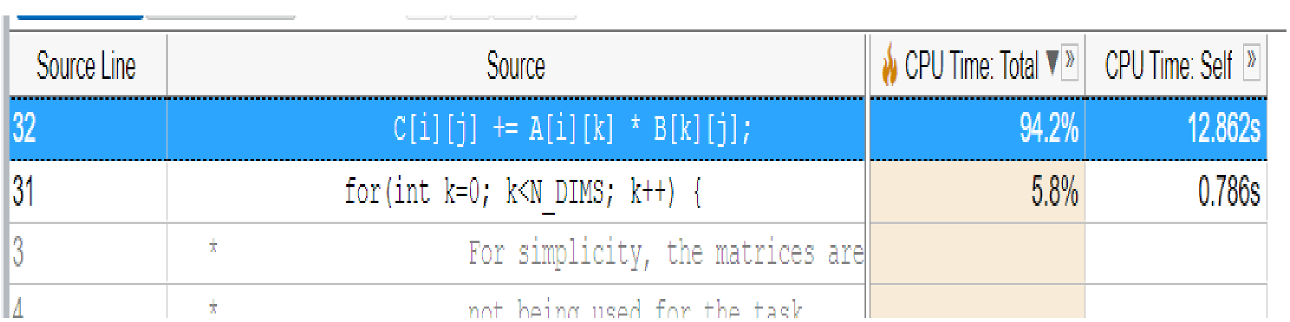
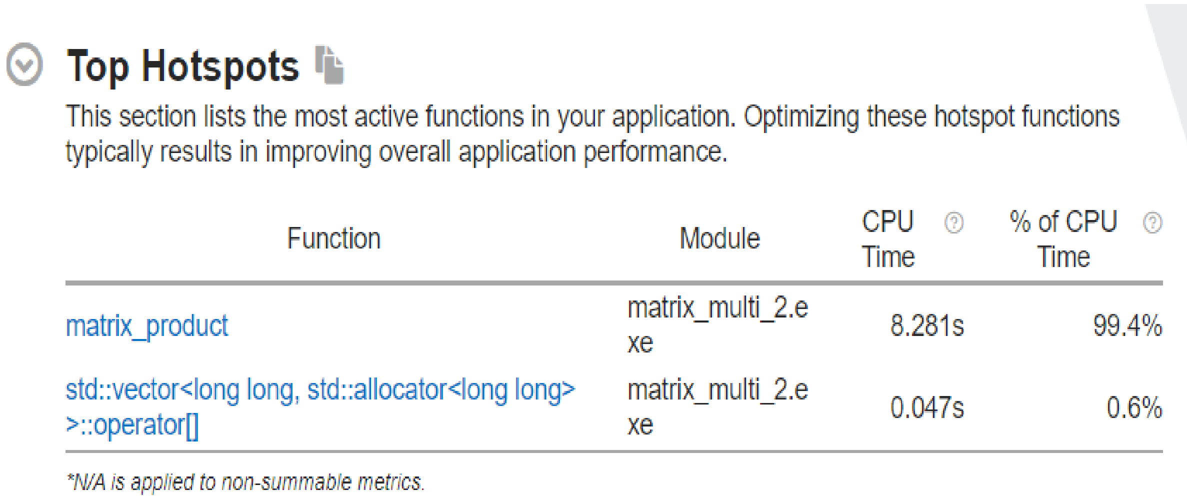


Figure 3.9: Matrix Product function - Hotspot

- **Matrix Multiplication-2** (`matrix_multi_2.cpp`) The top hotspots that were identified along with their CPU time can be found in the screenshot of hotspot test for the program as follows:



Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
matrix_product	matrix_multi_2.exe	8.281s	99.4%
std::vector<long long, std::allocator<long long> >::operator[]	matrix_multi_2.exe	0.047s	0.6%

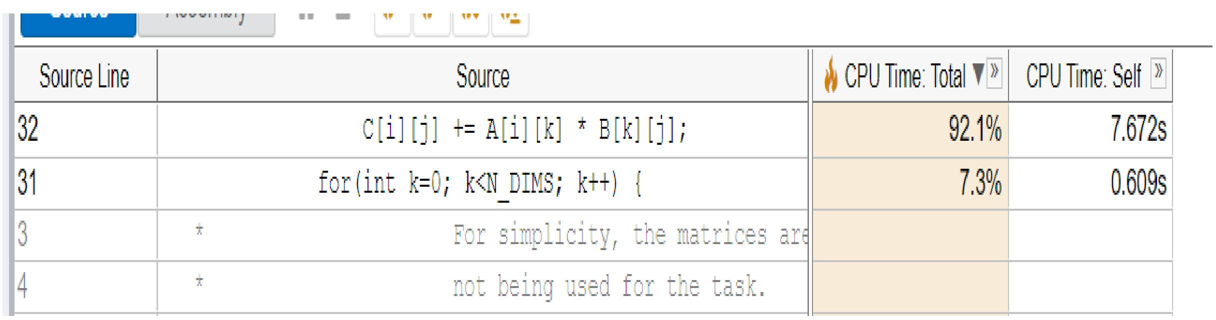
*N/A is applied to non-summable metrics.

Figure 3.10: Hotspots for 2nd Matrix Multiplication Algorithm

The statements in the program's source code that were responsible for consuming most of the CPU time (in descending order of the % of CPU time consumed) are as follows:

Line	Code	CPU Time (Total)	CPU Time (Self)
32	`C[i][j] += A[i][k]*B[k][j]`	92.1 %	7.672 s
31	`for(int k=0; k<N_DIMS; k++) {`	7.3 %	0.609 s

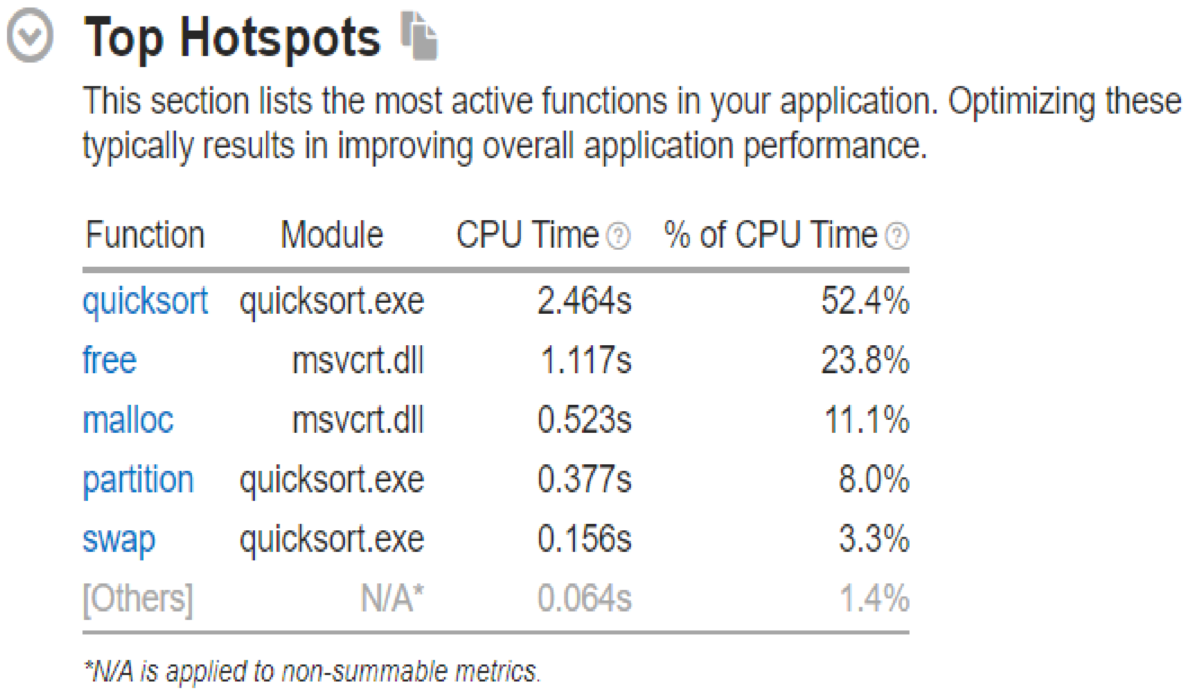
Screenshots supporting the above table are attached as follows:



Source Line	Source	CPU Time: Total	CPU Time: Self
32	`C[i][j] += A[i][k] * B[k][j];`	92.1%	7.672s
31	`for(int k=0; k<N_DIMS; k++) {`	7.3%	0.609s
3	`* For simplicity, the matrices are`		
4	`* not being used for the task.`		

Figure 3.11: Matrix Product function - Hotspot

- **QuickSort Algorithm** (`quicksort.cpp`) The top hotspots that were identified along with their CPU time can be found in the screenshot of hotspot test for the program as follows:



Top Hotspots

This section lists the most active functions in your application. Optimizing these typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
quicksort	quicksort.exe	2.464s	52.4%
free	msvcrt.dll	1.117s	23.8%
malloc	msvcrt.dll	0.523s	11.1%
partition	quicksort.exe	0.377s	8.0%
swap	quicksort.exe	0.156s	3.3%
[Others]	N/A*	0.064s	1.4%

*N/A is applied to non-summable metrics.

Figure 3.12: Hotspots for BFS Algorithm

The statements in the program's source code that were responsible for consuming most of the CPU time (in descending order of the % of CPU time consumed) are as follows:

Line	Code	CPU Time (Total)	CPU Time (Self)
45	`quicksort(nums, lo, p-1);`	52.4 %	0.208 s
44	`long p = partition(nums, lo, hi)`	32.7 %	1.538 s
46	`quicksort(nums, p+1, hi);`	15.3 %	0.717 s
33	`slow_ptr++;`	4.4 %	205.65 ms
22	`a = b;`	2.1 %	96.372 ms
31	`if(nums[i] < pivot) {`	1.9 %	91.401 ms
30	`for(long i=lo; i<hi; i++) {`	1.7 %	80.007 ms
23	`b = c;`	1.3 %	59.299 ms

Screenshots supporting the above table are attached as follows:

Source Line	Source	CPU Time: Total	CPU Time: Self
45	quicksort(nums, lo, p-1);	52.4%	0.208s
44	long p = partition(nums, lo, hi);	32.7%	1.538s
46	quicksort(nums, p+1, hi);	15.3%	0.717s
4	* NOTE: Increase the N_ELEM value for more comput		
5	* Higher values are preferred as you'll get		

Figure 3.13: quicksort function - Hotspot

Source Line	Source	CPU Time: Total	CPU Time: Self
33	slow_ptr++;	4.4%	205.650ms
31	if(nums[i] < pivot) {	1.9%	91.401ms
30	for(long i=lo; i<hi; i++) {	1.7%	80.007ms
3	*		
4	* NOTE: Increase the N_ELEM value for more computat		

Figure 3.14: partition function - Hotspot

Source Line	Source	CPU Time: Total	CPU Time: Self
22	a = b;	2.1%	96.372ms
23	b = c;	1.3%	59.299ms
1	/**		

Figure 3.15: swap function - Hotspot

3.3 Simulating With ChampSim

The traces were created using `pin` tool and `obj-intel64/champsim_tracer.so`, which was generated using `make_tracer.sh` script in the champsim directory. Also, the traces were compressed using `xz` tool. These traces were then stored in traces directory of champsim for simulation.

For all the cases below, champsim was build and run with the parameters given in the problem statement. (bimodal, no, no, no, no, lru, 1) for buliding and 10M + 10M instructions for running (warmup + simulation).

Note: For all the cases below, following data are summarized, **IPC(instruction per cycle)** for entire program , **MPKI(Misses per Kilo Instructions)** and **AML (Average Miss Latency)** for each cache level. MPKI values for each level were calculated from the simulation results (misses/(10^4)). The data is summarized in tabular format and the cache corresponding to data for MPKI and AML is also mentioned.

Note: Plots are attached after the tabular data for all the cases and programs, in Section 3.4.

1. **Baseline:** No change in `cache.h`.

   For the default values of caches in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/baseline` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.764871	0.687400	0.687241	0.409923
MPKI - L1D	2.1156	0.7515	0.7515	6.6413
MPKI - L1I	0.0012	0.0001	0.0001	0.0001
MPKI - L2C	1.8836	0.7445	0.7436	1.3843
MPKI - LLC	0.7879	0.7434	0.7434	1.3828
AML - L1D	68.9671	138.425	138.792	37.7478
AML - L1I	175.833	44	44	215
AML - L2C	60.7152	124.59	125.111	109.146
AML - LLC	73.4396	94.7659	95.1364	79.2318

2. **Direct Mapped Cache:** Changes in `cache.h`.

TLB cache values were kept as it is. Since direct mapping is 1-way, to keep size of cache fixed, the values were changed as follows, for L1 instruction cache, L1 data cache, L2 cache and Last Level Cache.

- Set value of `_SET` for all four caches listed above to **old set value * old ways value**
- Set value of `_WAY` for all four caches listed above to **1**.

For the above changes in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/direct-mapped` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.763515	0.683596	0.683784	0.297432
MPKI - L1D	2.8124	1.4203	1.4326	128.5648
MPKI - L1I	7.6703	0.9609	0.9443	0.0140
MPKI - L2C	1.7744	0.7899	0.7968	1.4961
MPKI - LLC	0.8432	0.7703	0.7619	2.2886
AML - L1D	55.3108	81.8068	82.9345	11.9727
AML - L1I	14.3639	14.3404	14.5849	67.5143
AML - L2C	64.319	120.241	122.775	143.698
AML - LLC	73.8272	94.0328	98.6354	74.6566

Comments: Overall, IPC **decreases** slightly when compared to the baseline values. In general, average miss latencies decreases and the MPKI values increases significantly resulting in degradation of MPKI. Since, we decreased the number of ways to 1, hence decrease in associativity resulted in increase in conflict misses drastically (since, all lines are individual sets now), which inturn increases the MPKI, and a slight decrease in the IPC values. Also, the decrease in average miss latencies can be attributed to the decrease in number of comparators which can lead to faster data load on miss.

For cache level L1I and L1D, MPKI increases significantly compared to the baseline values. This can be due to the fact that on decreasing associativity, the conflict misses for instructions drastically increases, hence the MPKI values for level one cache increases sharply. One can refer to the plots for better visualization of results.

3. **Fully Associative Cache:** Changes in `cache.h`.

TLB cache values were kept as it is. Since fully associative is 1 set cache, to keep size of cache fixed, the values were changed as follows, for L1 instruction cache, L1 data cache, L2 cache and Last Level Cache.

- Set value of `_SET` for all four caches listed above to **1**.
- Set value of `_WAY` for all four caches listed above to **old set value * old ways value**

For the above changes in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/fully-associative` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.764314	0.687400	0.687241	0.409919
MPKI - L1D	2.1158	0.7515	0.7515	6.6413
MPKI - L1I	0.0012	0.0001	0.0001	0.0001
MPKI - L2C	1.9321	0.7436	0.7436	1.3843
MPKI - LLC	0.7879	0.7434	0.7434	1.3833
AML - L1D	69.32058	138.425	138.785	37.9418
AML - L1I	175.833	44	44	44
AML - L2C	59.5841	124.74	125.104	110.064
AML - LLC	72.5525	94.7659	95.1299	80.1222

Comments: Overall, all the metrics i.e IPC, MPKI and average miss latecies are roughly same for fully associative case as compared to the baseline case. One can refer to plots for more visulization.

Since, the default values were n-way associative for baseline data in `cache.h`, increasing associativity would have decreased conflict misses, but here, it is likely that after a certain k-way associativity, the conflict misses reduction is optimal and is further not possible. Hence, MPKI is similar to that of the baseline condition. The same reasoning can be attributed to IPC and Miss latencies.

4. **Reduced Size Cache:** Changes in `cache.h`.

TLB cache values were kept as it is. To reduce the cache size we can half the set size, hence the values were changed as follows, for L1 instruction cache, L1 data cache, L2 cache and Last Level Cache.

Also, since we are changing the size of cache, we will also need to consider new latencies for cache levels. The new latencies were calculated using tool `CACTI`, and set accordingly for different cache levels.

- Set value of `_SET` for all four caches listed above to **old value/2**.
- Set value of `_LATENCY` for L2 Cache level to **9**.
- Set value of `_LATENCY` for Last Cache level to **18**.
- Keep latencies as default values for other levels.

For the modified values in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/reduced-size` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.741176	0.687309	0.687169	0.410574
MPKI - L1D	2.1354	0.7520	0.7520	6.6413
MPKI - L1I	0.0029	0.0139	0.0139	0.0001
MPKI - L2C	1.9891	0.7515	0.7513	1.3851
MPKI - LLC	1.054	0.7435	0.7434	1.3837
AML - L1D	98.89	147.909	148.773	49.6357
AML - L1I	168.724	21.7914	20.6043	209
AML - L2C	91.3586	134.16	135.038	170.877
AML - LLC	121.462	108.418	109.303	144.023

Comments: Overall, IPC remains roughly same (slightly smaller for some cases) as compared to the baseline values. In general, average miss latencies increases and the MPKI values increases, resulting in degradation of MPKI. Since, we decreased the size of cache by half, the **capacity** and **conflict** misses will increase resulting in increase in MPKI values.

Also, the hit time will decrease and since we have smaller cache size than earlier, the chances of hit in earlier level of caches decreases, hence average miss latency in cycles increases.

5. **Doubled Size Cache:** Changes in `cache.h`.

TLB cache values were kept as it is. To double the cache size we can double the set size, hence the values were changed as follows, for L1 instruction cache, L1 data cache, L2 cache and Last Level Cache.

Also, since we are changing the size of cache, we will also need to consider new latencies for cache levels. The new latencies were calculated using tool `CACTI`, and set accordingly for different cache levels.

- Set value of `_SET` for all four caches listed above to **2*old value**.
- Set value of `_LATENCY` for L2 Cache level to **13**.
- Set value of `_LATENCY` for Last Cache level to **24**.
- Keep latencies as default values for other levels.

For the modified values in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/doubled-size` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.772407	0.687375	0.687212	0.407948
MPKI - L1D	2.0821	0.7505	0.7506	6.5665
MPKI - L1I	0.0012	0.0001	0.0001	0.0
MPKI - L2C	1.1078	0.7435	0.7434	1.3837
MPKI - LLC	0.7879	0.7434	0.7434	1.3826
AML - L1D	66.5506	145.233	145.706	41.9894
AML - L1I	189.5	54	17	NAN
AML - L2C	91.4346	128.435	128.943	113.841
AML - LLC	76.5452	91.4477	91.9431	76.9022

Comments: Overall, IPC remains roughly same (slightly larger for some cases) as compared to the baseline values. In general, average miss latencies decreases and the MPKI values decreases, resulting in improvement of MPKI. Since, we increases the size of cache by a factor of two, the **capacity** and **conflict** misses will decrease resulting in decrease in MPKI values.

Also, the hit time will increase and since we have larger cache size than earlier, the chances of hit in earlier level of caches increases, hence average miss latency in cycles decreases.

6. **Doubled MSHR Cache:** Changes in `cache.h`.

TLB cache values were kept as it is. To double the size of MSHR, we can double the values for corresponding fields in `cache.h`, hence, the values were changed as follows, for L1 instruction cache, L1 data cache, L2 cache and Last Level Cache.

- Set value of `_MSHR_SIZE` for all four caches listed above to **2*old value**.

For the above changes in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/doubled-mshr` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.764876	0.687401	0.687254	0.409923
MPKI - L1D	2.1156	0.7515	0.7515	6.6413
MPKI - L1I	0.0012	0.0001	0.0001	0.0001
MPKI - L2C	1.8826	0.7445	0.7436	1.3843
MPKI - LLC	0.7879	0.7434	0.7434	1.3828
AML - L1D	68.9989	229.279	229.735	37.7478
AML - L1I	175.833	44	44	215
AML - L2C	60.751	216.297	217.021	109.146
AML - LLC	73.5251	186.609	187.071	79.2318

Comments: Overall, the metrics IPC and MPKI are almost same for this case as compared to the baseline case. One can refer to plots for more visulization. For the case of `quicksort.cpp` though, the MPKI value has improved, and the new value was observed 0.

Since, the misses are same, earlier it was handled by half sized MSHR registers, on increasing size, there is no practical improvement in IPC and MPKI. This effect can be attributed to no difference in cache size and the internal schema of data mapping. Although, increasing MSHR size might have led to increase in time after miss (that is there will be more waiting for misses after acertain miss) which led to increase in average miss latency (in cycles) for some datapoints.

7. **Reduced MSHR Cache:** Changes in `cache.h`.

TLB cache values were kept as it is. To reduce the size of MSHR, we can half the values for corresponding fields in `cache.h`, hence, the values were changed as follows, for L1 instruction cache, L1 data cache, L2 cache and Last Level Cache.

- Set value of `_MSHR_SIZE` for all four caches listed above to **old value/2**.

For the above changes in `ChampSim/inc/cache.h`, the following values were observed and also stored in `/reduced-mshr` directory.

Metric	`bfs.cpp`	`matrix_multi.cpp`	`matrix_multi_2.cpp`	`quicksort.cpp`
IPC	0.764862	0.687028	0.686792	0.409923
MPKI - L1D	2.1156	0.7515	0.7515	6.6413
MPKI - L1I	0.0012	0.0001	0.0001	0.0001
MPKI - L2C	1.8826	0.7445	0.7436	1.3843
MPKI - LLC	0.7879	0.7434	0.7434	1.3828
AML - L1D	68.9387	120.389	120.767	37.7478
AML - L1I	175.833	44	44	215
AML - L2C	60.6837	106.384	106.895	109.146
AML - LLC	73.3643	76.5338	76.9154	79.2318

Comments: Overall, the metrics IPC and MPKI are almost same for this case as compared to the baseline case. One can refer to plots for more visulization.

Since, the misses are same, earlier it was handled by double sized MSHR registers, on decreasing size, there is no practical improvement in IPC and MPKI. This effect can be attributed to no difference in cache size and the internal schema of data mapping. Although, decreasing MSHR size might have led to decrease in time after miss (since, now it will wait for less time), which led to decrease in average miss latency (in cycles) for some datapoints.

3.4 Plots (ChampSim Simulation)

1. **Instructions Per Cycle For all the Cases**

   Normalised Instruction was calculated for an IPC by dividing it with corresponding baseline IPC. ($IPC_n = IPC/Base\ IPC$)

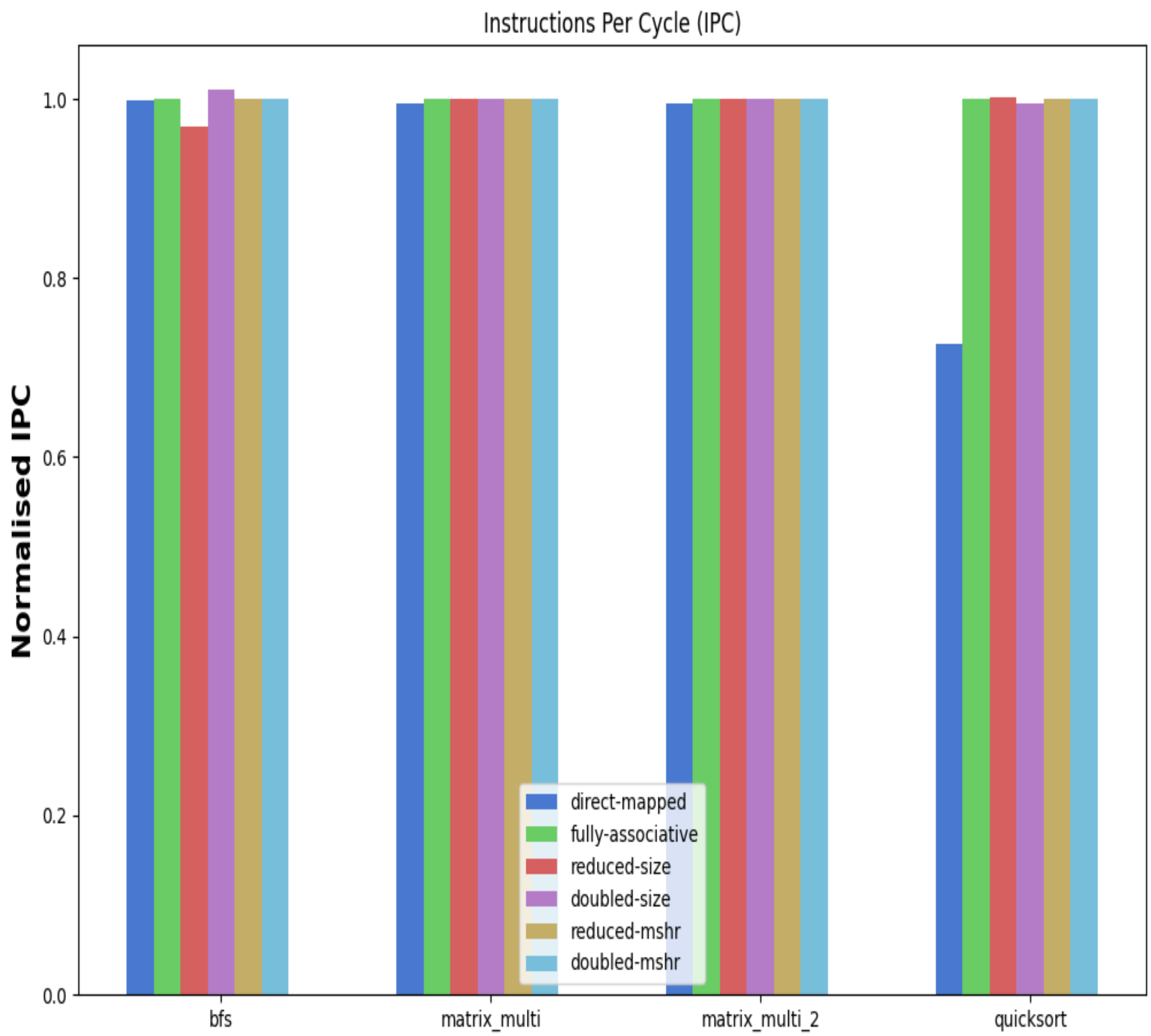


Figure 3.16: Instructions Per Cycle For all Programs and Cases

Note: For each MPKI value, MPKI degradation percentage was calculated as $[(MPKI - Base\ MPKI)/(Base\ MPKI)] * 100$.

2. **MPKI Comparison Level 1 Data Cache**

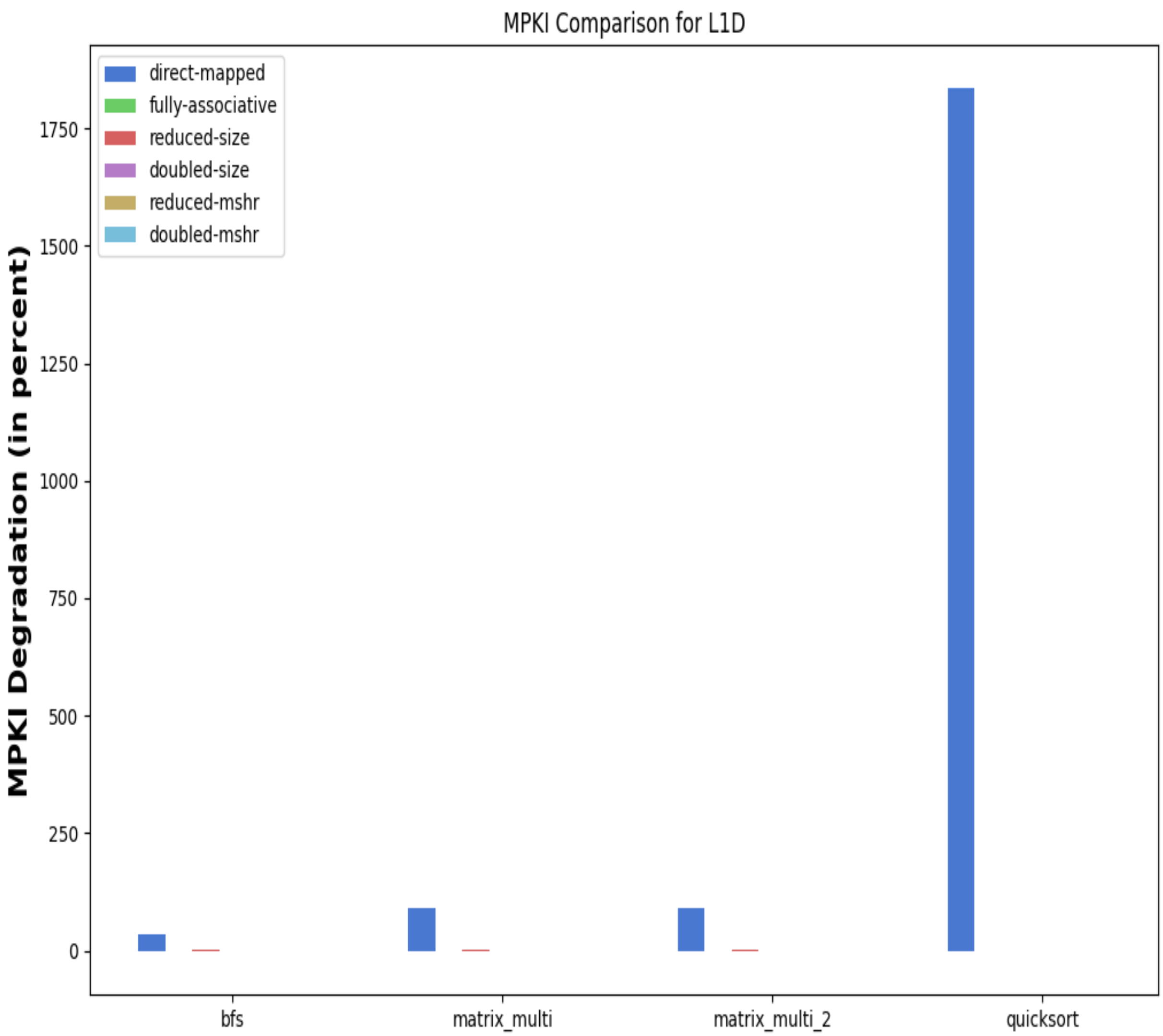


Figure 3.17: MPKI Comparison for L1D

3. **MPKI Comparison Level 1 Instruction Cache**

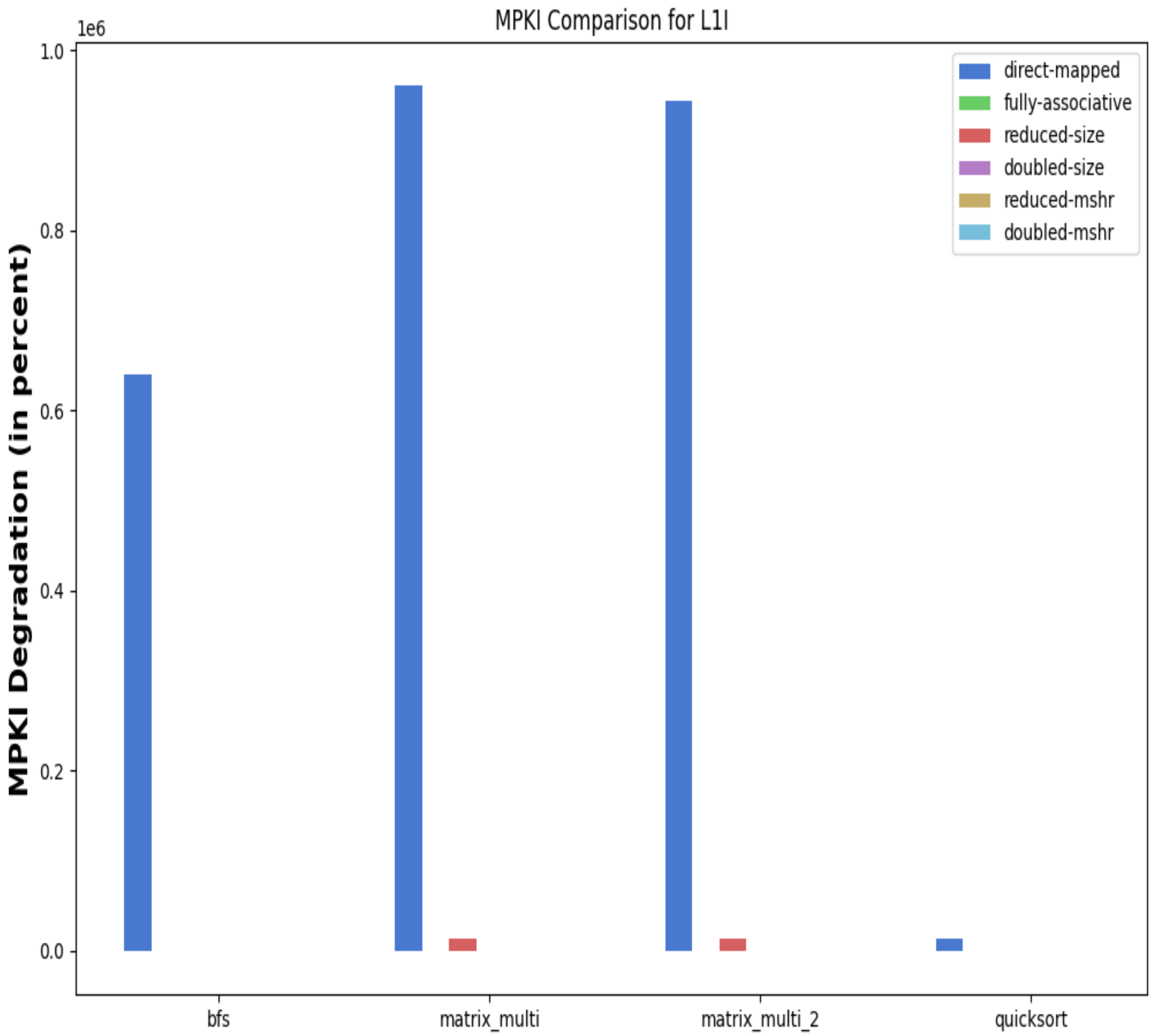


Figure 3.18: MPKI Comparison for L1I

4. **MPKI Comparison Level 2 Cache**

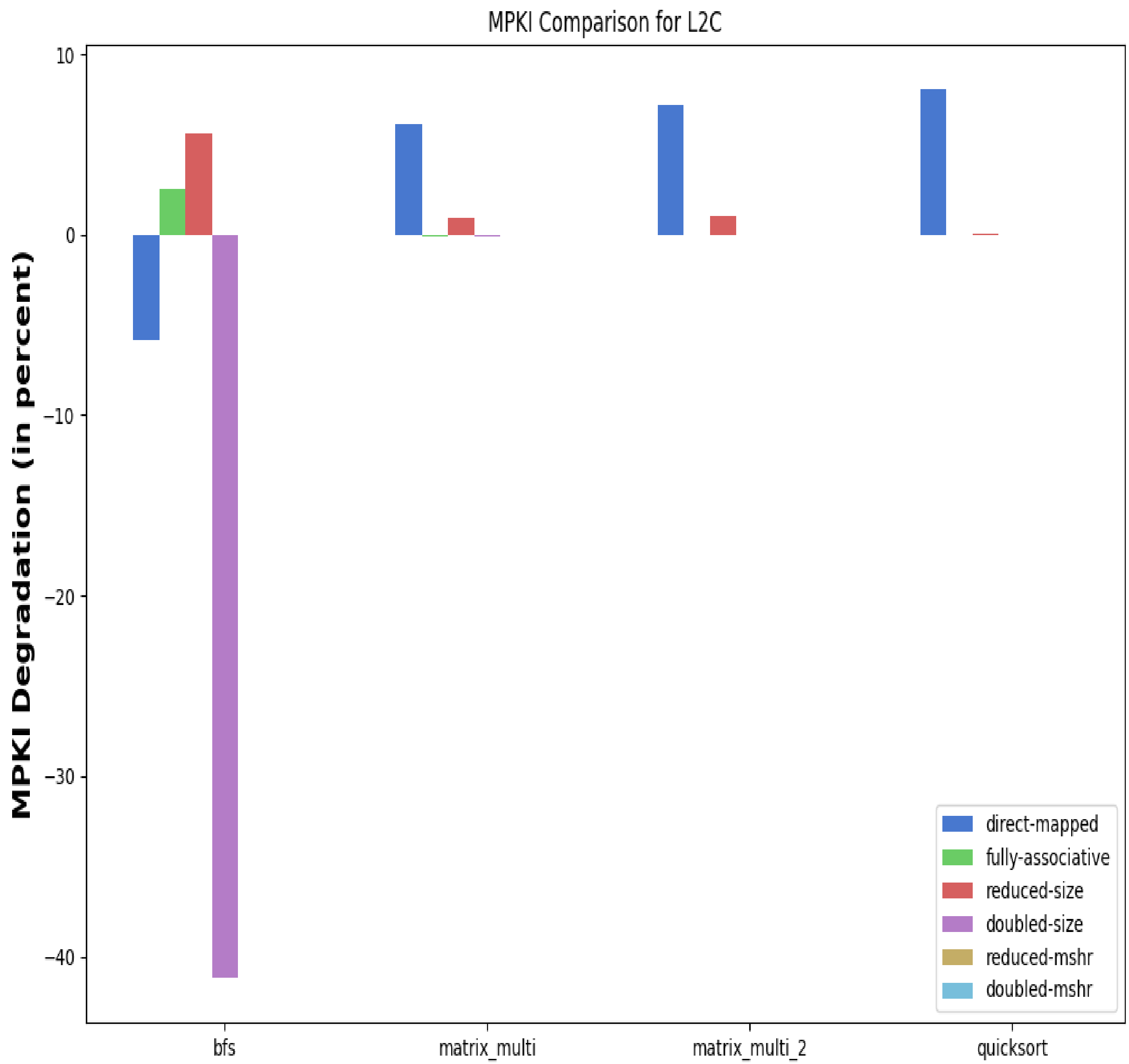


Figure 3.19: MPKI Comparison for L2C

5. **MPKI Comparison Last Level Cache**

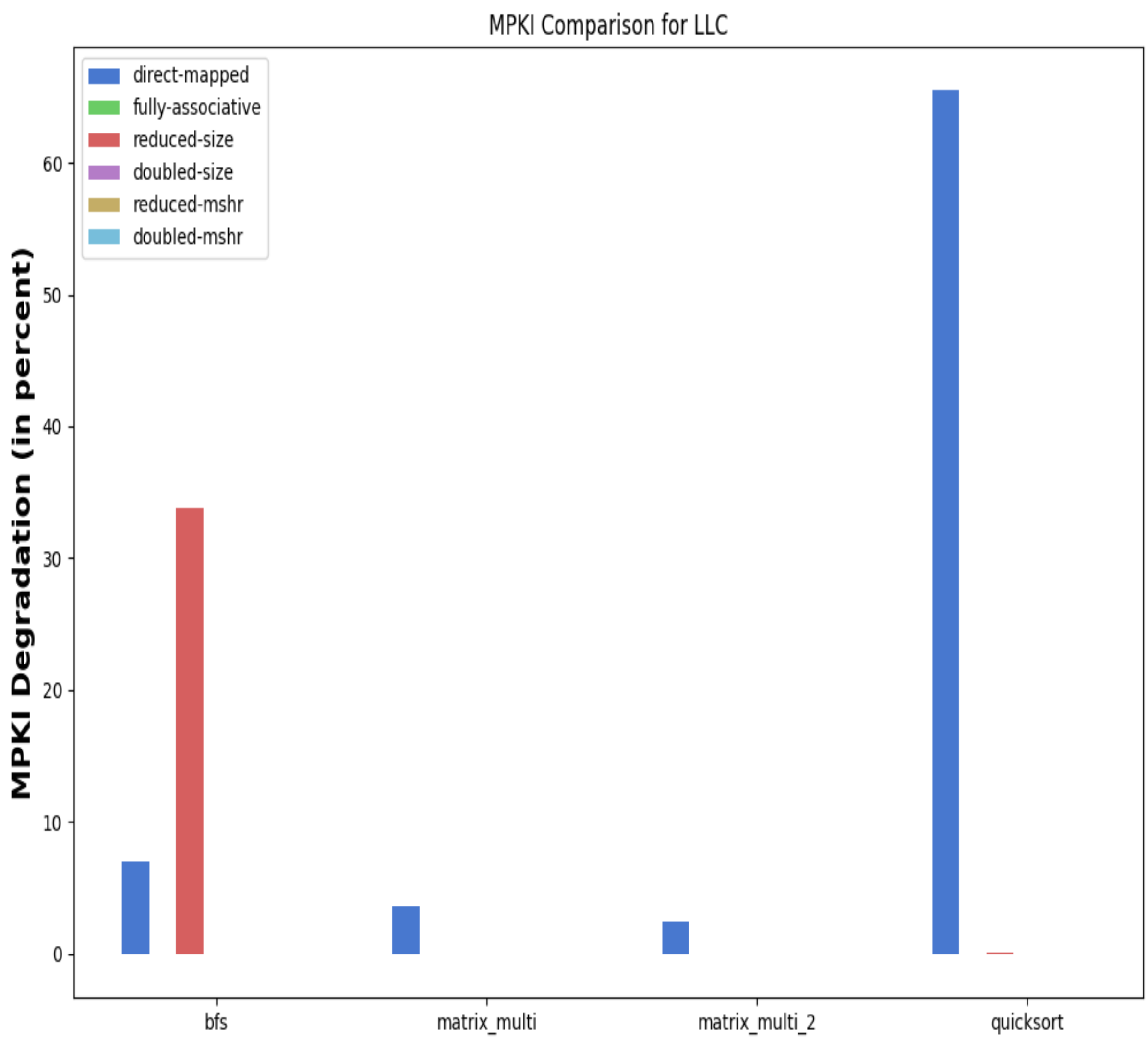


Figure 3.20: MPKI Comparison for LLC