CS341: Computer Architecture Lab

Lab 4: Intel VTune Profiler and ChampSim

Report

Richeek Das - 190260036

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 explore profiling and analysing the runtime 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.

1. Part 0: Getting Things Ready

1.1 Install Intel VTune Profiler

Done!

1.2 Challenges faced with VTune profiler

The overall process was very simple. Initially I found out it won't work on my machine since it has an **AMD Ryzen 5 5500u** CPU. I had to switch to an Intel based machine, **Intel i3 7100u**. Other than that, it was just downloading an application from the linked website and running it as administrator to check out the hardware profiles.

1.3 Docker installation

I had already installed docker on my Ubuntu machine with **AMD Ryzen 5 5500u**. I discovered **Intel's Pin Utility** works fine with most Ryzen CPUs and fortunately it worked with mine too. So I used ChampSim on a machine with Ubuntu 20.04 OS and AMD Ryzen 5 5500u CPU.

2. Part 1: Profiling with VTune

We work with VTunes and tabulate the following quantities:

Performance Snapshot:

1. Observed IPC for each program

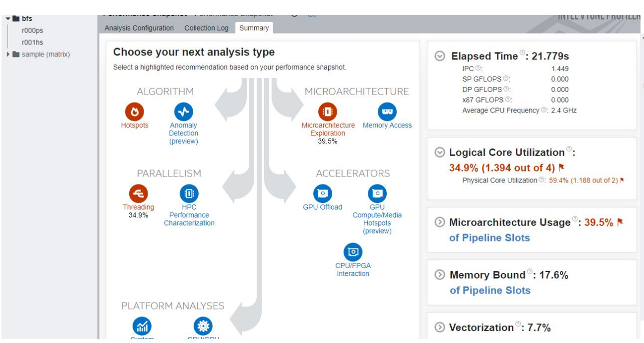
bfs	matrix_multi	matrix_multi_2	quicksort
1.449	0.808	0.881	0.984

2. Observed logical core utilization and physical core utilization for each program

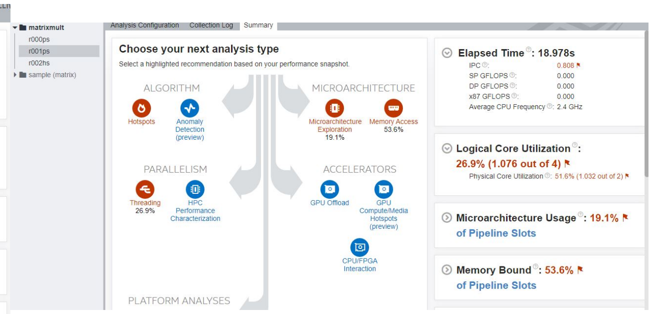
	bfs	matrix_multi	matrix_multi_2	quicksort
logical core	34.9% (1.394/4)	26.9% (1.076/4)	27.1% (1.085/4)	58.8% (2.350/4)
physical core	59.4% (1.188/2)	51.6% (1.032/2)	52.3% (1.047/2)	79.2% (1.585/2)

3. Observed % of the pipeline slots are memory bound for each program

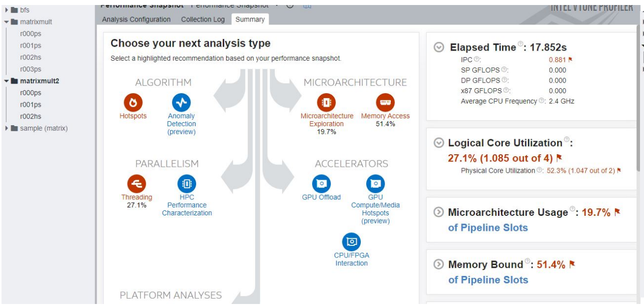
bfs	matrix_multi	matrix_multi_2	quicksort
17.6%	53.6%	51.4%	17.0%



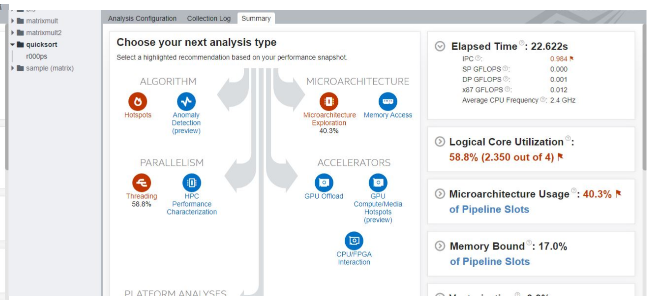
bfs



matrix_multi



matrix_multi_2



quicksort

Hotspots:

1: The top hotspots that were identified along with their CPU time.

bfs:

Table 2.1: Most intensive functions along with their CPU time. **Note that we see cygwin1.dll here, because the code has been compiled using the cygwin port.**

Function	Module	CPU Time
bfs	bfs.exe	7.047s
main	bfs.exe	3.261s
func@0x180178f10	cygwin1.dll	2.818s

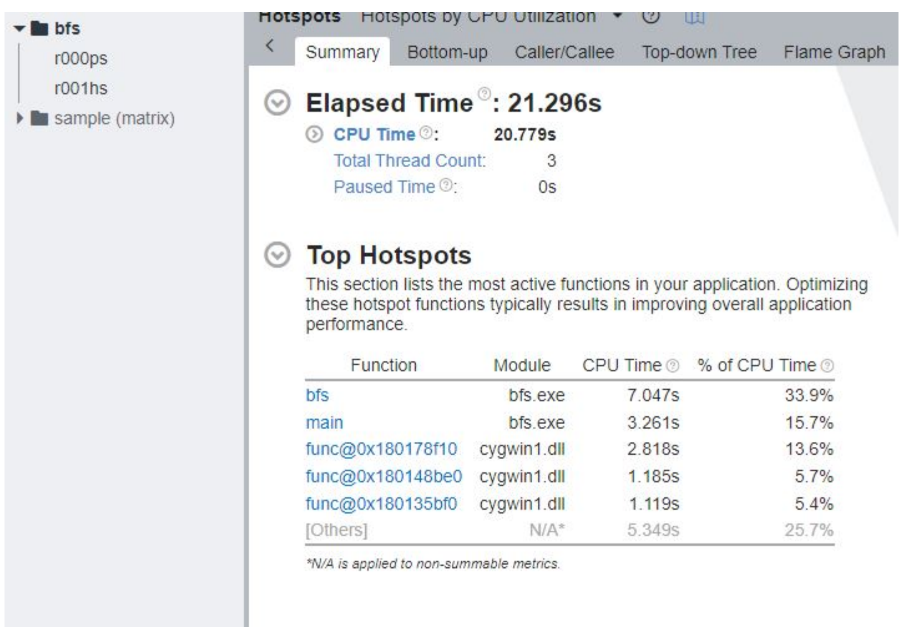


Figure 2.3: Hotspots of **bfs**

matrix_multi.cpp:

Function	Module	CPU Time
matrix_product	matrix_multi.exe	18.919s (99.1%)
LoadLibraryExW	KERNBASE.dll	0.148s
rest in	screenshot	<1%

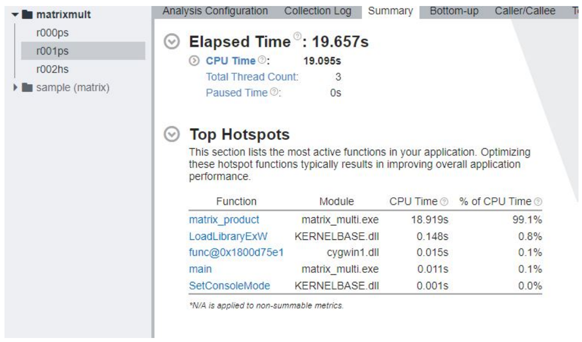
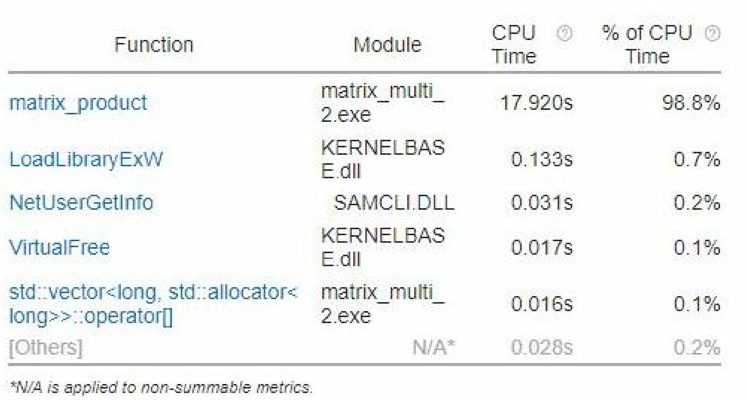


Figure 2.4: Hotspots of **matrix_multi**

matrix_multi_2.cpp:

Function	Module	CPU Time
matrix_product	matrix_multi_2.exe	17.920s (98.8%)
LoadLibraryExW	KERNBASE.dll	0.133s
rest in	screenshot	<1%



Function	Module	CPU Time	% of CPU Time
matrix_product	matrix_multi_2.exe	17.920s	98.8%
LoadLibraryExW	KERNELBASE.dll	0.133s	0.7%
NetUserGetInfo	SAMCLI.DLL	0.031s	0.2%
VirtualFree	KERNELBASE.dll	0.017s	0.1%
std::vector<long, std::allocator< long>>::operator[]	matrix_multi_2.exe	0.016s	0.1%
[Others]	N/A*	0.028s	0.2%

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

Figure 2.5: Hotspots of **matrix_multi_2**

quicksort.cpp:

Function	Module	CPU Time
quicksort	quicksort.exe	20.446
func@0x140405257	ntoskrnl.exe	12.683s
func@0x1800d75e1	cygwin1.dll	10.785s
func@0x140246b50	ntoskrnl.exe	8.876s
partition	quicksort.exe	5.933s

Top Hotspots

This section lists the most active functions in your applicat
results in improving overall application performance.

Function	Module	CPU Time
quicksort	quicksort.exe	20.446s
func@0x140405257	ntoskrnl.exe	12.683s
func@0x1800d75e1	cygwin1.dll	10.785s
func@0x140246b50	ntoskrnl.exe	8.876s
partition	quicksort.exe	5.933s
[Others]	N/A*	34.986s

Figure 2.6: Hotspots of **quicksort**

2: 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).

bfs:

`src code`	CPU Time
`line 97: right_child=curr_node->right;`	20.6%
`line 120: bfs(root)`	15.7%
`line 96: left_child=curr_node->left;`	7.5%
`line 92: for(int i=0; i<q_size; i++) {`	3.1%
`line 100: if(left_child) node_Q.push(left_child);`	1.8%
`rest`	individually <1%

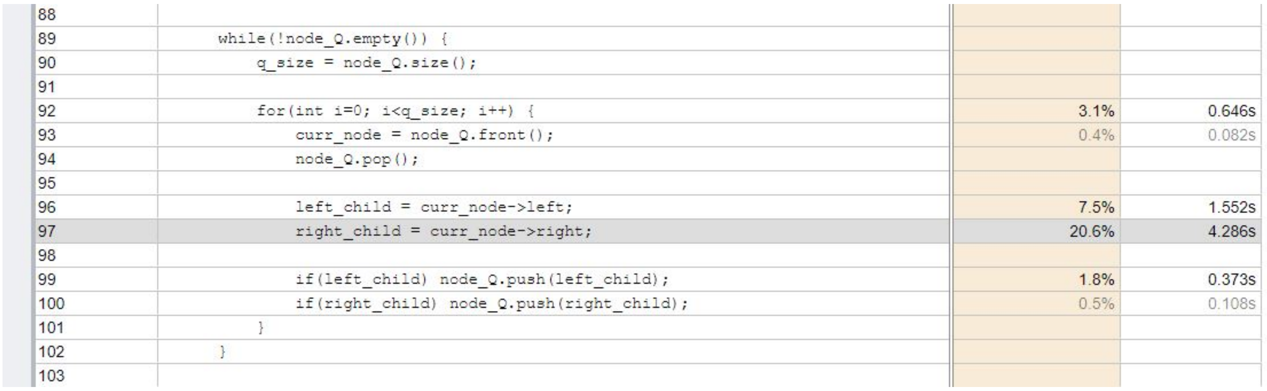


Figure 2.7: Time consuming statements of **bfs**

matrix_multi.cpp:

src code	CPU Time
`line 32:  C[i][j] += A[i][k] * B[k][j];`	90.6%
`line 31:  for(int k=0; k<N_DIMS; k++) {`	8.5%
`rest`	<1%

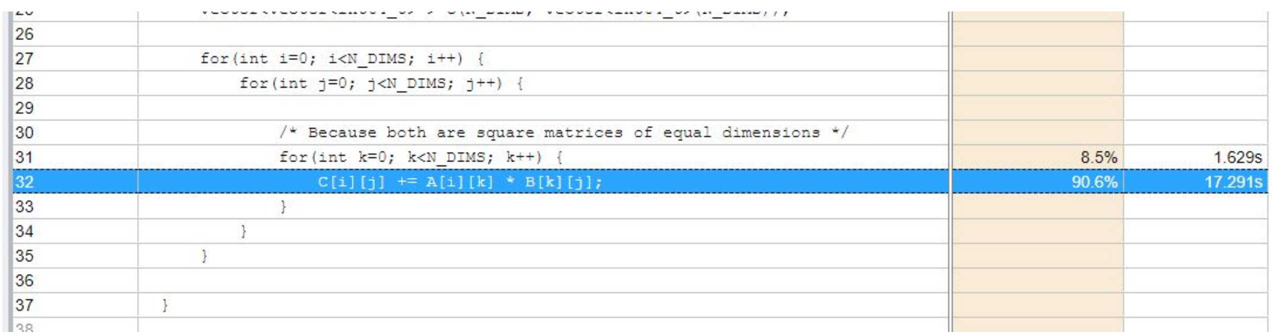


Figure 2.8: Time consuming statements of **matrix_multi**

matrix_multi_2.cpp:

src code	CPU Time
`line 32:  C[i][j] += A[i][k] * B[k][j];`	89.3%
`line 31:  for(int k=0; k<N_DIMS; k++) {`	9.4%
`rest`	<1%

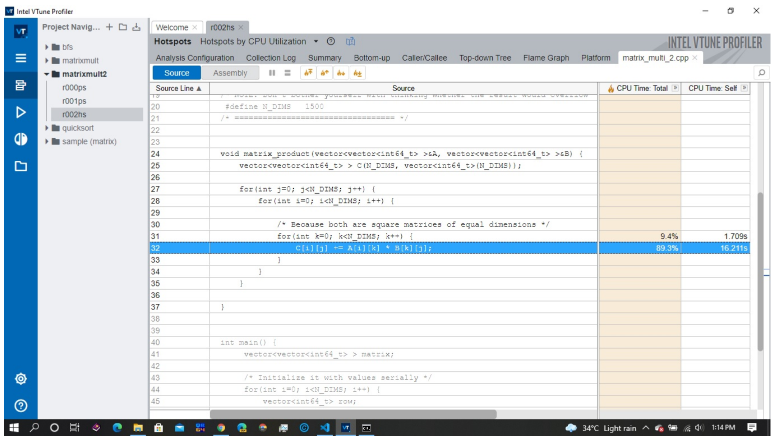
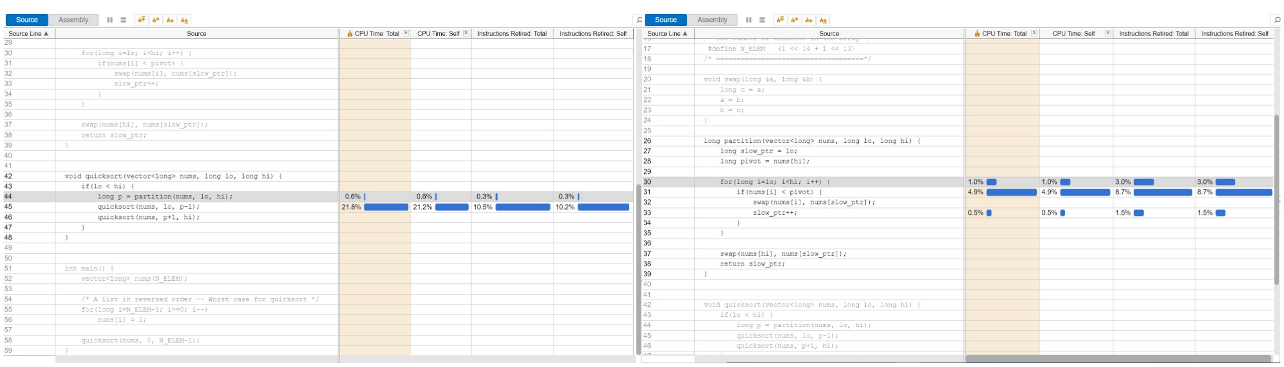


Figure 2.9: Time consuming statements of **matrix_multi_2**

quicksort.cpp:

src code	CPU Time
`line 45: quicksort(nums, lo, p-1)`	21.8%
`line 31: if(nums[i] < pivot) {`	4.9%
`line 30: for(long i=lo; i<hi; i++) {`	1.0%
`rest`	<1%



(a) (b)

Figure 2.10: Time consuming statements of **quicksort**

3. Part 2: Simulating with ChampSim

We have already prepared *traces* for each program. Here are the baseline results:

3.1 Baseline

Table 3.1: IPC of baseline

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.844404	0.684276	0.683893	0.794884

Table 3.2: MPKI for each of the programs at baseline configuration

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	5.571	1.859	1.857	15.415
L1I	0.000	0.000	0.000	0.001
L2C	487.948	496.027	495.775	133.602
LLC	326.452	552.554	552.904	516.793

Table 3.3: Average Miss Latency for each of the programs at baseline configuration

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	93.327	139.311	140.155	87.700
L1I	44.000	44.000	44.000	215.000
L2C	80.441	124.927	125.808	333.852
LLC	77.340	94.028	95.066	320.067

Note: We will use ↓ and ↑ to denote the decrease/increase in IPC wrt to this baseline, in the later sections.

3.2 Effect of using Direct-Mapped Cache at all levels

Table 3.4: IPC of Direct-Mapped Cashe case. **IPC did not improve in any of the tested programs**

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.842032 ↓	0.678711 ↓	0.680293 ↓	0.792479 ↓

Table 3.5: MPKI for each of the programs.

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	8.224	4.840	4.998	16.276
L1I	55.085	2.761	0.690	0.021
L2C	114.121	177.166	202.999	255.872
LLC	345.944	565.472	562.338	291.973

Table 3.6: Average Miss Latency for each of the programs

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	71.210	64.075	69.090	67.130
L1I	14.023	14.426	16.105	149.660
L2C	75.789	124.516	138.884	125.932
LLC	79.158	96.949	112.664	195.521

Observations:

- We see a decrease in IPC for each of the programs. This is obvious due the increase in conflict misses (no associativity).
- We see a rise in MPKI due to the rise in conflict misses. But L2C sees a decrease in MPKI for bfs, matrix_multi, matrix_multi_2.
- We see drop in the Average Miss Latency for each cache and program since we now need lower comparator latency for implementing zero associativity.

3.3 Effect of using Fully-Associative Cache at all levels

Table 3.7: IPC of fully-associative cache case. **IPC improved only for matrix_multi_2**

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.844037 ↓	0.684264 ↓	0.684298 ↑	0.792999 ↓

Table 3.8: MPKI for each of the programs.

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	5.571	1.857	1.857	15.413
L1I	0.000	0.000	0.000	0.000
L2C	495.351	495.775	495.775	377.633
LLC	321.577	549.231	549.231	252.771

Table 3.9: Average Miss Latency for each of the programs

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	93.425	139.217	137.924	96.871
L1I	44.000	44.000	44.000	215.000
L2C	79.338	124.932	123.541	134.091
LLC	76.791	94.065	92.720	312.820

Observations:

- We see the IPC improved only for matrix_multi_2. This is particularly because of the fact, the baseline is associative enough to handle the conflict misses to a large degree. Increasing associativity increases the latency, which counteracts the additional conflict misses it resolves.
- We see almost similar values of MPKI in the fully-associative case and baseline case. This is particularly because of the fact, the baseline is associative enough to handle the conflict misses to a large degree.
- We see a slight increase in the Average Miss Latency for each cache and program since we now need higher comparator latency for implementing zero associativity.

3.4 Effect of halving the size of the caches at all levels

Table 3.10: IPC of the halved cache size case. **IPC only improved for matrix_multi_2**

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.835378 $\downarrow$	0.684034 $\downarrow$	0.683902 $\uparrow$	0.766297 $\downarrow$

Table 3.11: MPKI for each of the programs.

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	5.578	3.343	3.365	15.658
L1I	0.004	0.056	0.055	0.000
L2C	499.054	279.903	277.880	508.616
LLC	345.680	499.873	500.223	161.333

Table 3.12: Average Miss Latency for each of the programs

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	133.727	86.943	86.779	150.921
L1I	81.800	20.335	19.352	211.000
L2C	119.255	128.149	128.715	165.567
LLC	132.133	101.778	102.134	572.539

Latency Calculation:

L1I	**L1D**	**L2C**	**LLC**
4	5	10	18

For calculating the latency we used **CACTI**. We lookup the access times and clock cycle times for the different cache parameters in the problem statement: `./cacti -infile cache.cfg | grep time`. We find the following quantity:

$$\frac{\text{access time}_{case}/\text{clock cycle time}_{case}}{\text{access time}_{baseline}/\text{clock cycle time}_{baseline}}$$

We scale the LATENCY constants defined in `cache.h` file according to this ratio and take its `ceil()`.

Observations:

- We see the IPC improved only for matrix_multi_2.
- We see an increase in MPKI for L1D in matrix_multi and matrix_multi_2. This is particularly expected since both of these programs are memory intensive. Halving the cache size leads to decrease in cache hits and increased MPKI.
- We see a drop in Average Miss Latency for matrix_multi and matrix_multi_2, but and increase for the rest.

3.5 Effect of doubling the size of the caches at all levels

Table 3.13: IPC of the doubled cache size case. **IPC did not improve in any of the tested programs**

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.789770 ↓	0.650507 ↓	0.650657 ↓	0.755001 ↓

Table 3.14: MPKI for each of the programs.

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	5.574	1.860	1.859	14.807
L1I	0.000	0.000	0.000	0.000
L2C	335.479	495.490	495.427	130.429
LLC	582.822	873.443	876.367	500.611

Table 3.15: Average Miss Latency for each of the programs

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	89.189	141.624	140.067	45.954
L1I	51.000	51.000	16.000	nan
L2C	107.827	125.343	123.726	133.050
LLC	76.727	89.595	88.058	89.451

Latency Calculation:

L1I	L1D	L2C	LLC
5	6	11	24

Calculations follow the same strategy used in the cache halved case above.

Observations:

- We see the IPC decreased for all the programs. This is because of the increase cache access time due to the increased cache size.
- We see an overall decrease in MPKI since there's increased cache hit because of increased cache size. But there is an increase in MPKI for LLC.
- We see almost similar Average Miss Latency in this case.

3.6 Effect of doubling the number of the MSHRs at all levels

Table 3.16: IPC of the doubled MSHR case. **IPC improved only for quicksort**

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.844402 ↓	0.682737 ↓	0.683352 ↓	0.796453 ↑

Table 3.17: MPKI for each of the programs.

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	5.571	1.859	1.889	15.421
L1I	0.000	0.000	0.000	0.000
L2C	487.996	496.090	495.712	133.543
LLC	326.420	553.371	552.982	516.647

Table 3.18: Average Miss Latency for each of the programs

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	93.381	242.980	245.139	154.327
L1I	44.000	44.000	44.000	215.000
L2C	80.487	228.896	231.175	645.292
LLC	77.416	198.182	200.473	653.846

Observations:

- We see an increase in IPC only in the case of quicksort. This might be due to MSHR helping in tackling cache misses in case of recursive code in quicksort.
- We don't see any effect on MPKI with doubling the number of MSHR at all levels.
- We see an increase in miss latency for the matrix_multi programs in the L1D, L2C, LLC caches (close to twice the original latency).

3.7 Effect of halving the number of MSHRs at all levels

Table 3.19: IPC of the reduced MSHR case. **IPC did not improve in any of the tested programs**

bfs	**matrix_multi**	**matrix_multi_2**	**quicksort**
0.844328 ↓	0.683079 ↓	0.681273 ↓	0.783982 ↓

Table 3.20: MPKI for each of the programs.

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	5.626	1.890	1.857	15.427
L1I	0.000	0.000	0.000	0.000
L2C	487.948	496.027	495.775	133.564
LLC	326.452	552.670	551.545	516.625

Table 3.21: Average Miss Latency for each of the programs

Cache level	**bfs**	**matrix_multi**	**matrix_multi_2**	**quicksort**
L1D	93.316	121.471	120.948	70.017
L1I	44.000	44.000	44.000	215.000
L2C	80.430	107.248	106.699	254.361
LLC	77.323	76.401	75.977	239.819

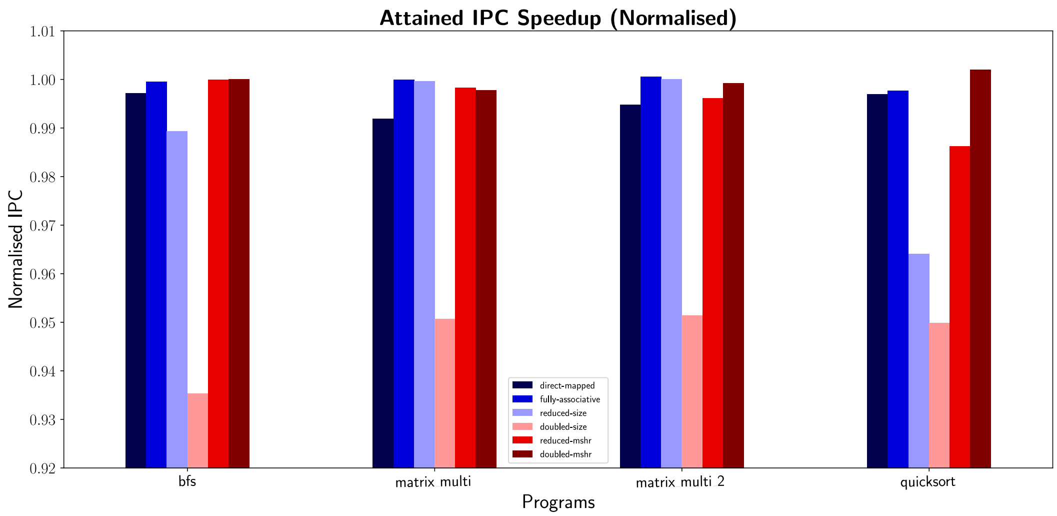
Observations:

- We see a decrease in IPC for each of the programs. This is expected considering the reduction in concurrency of serving memory cache misses with reduction in the number of MSHRs.
- We don't see any change in the MPKI for the programs.

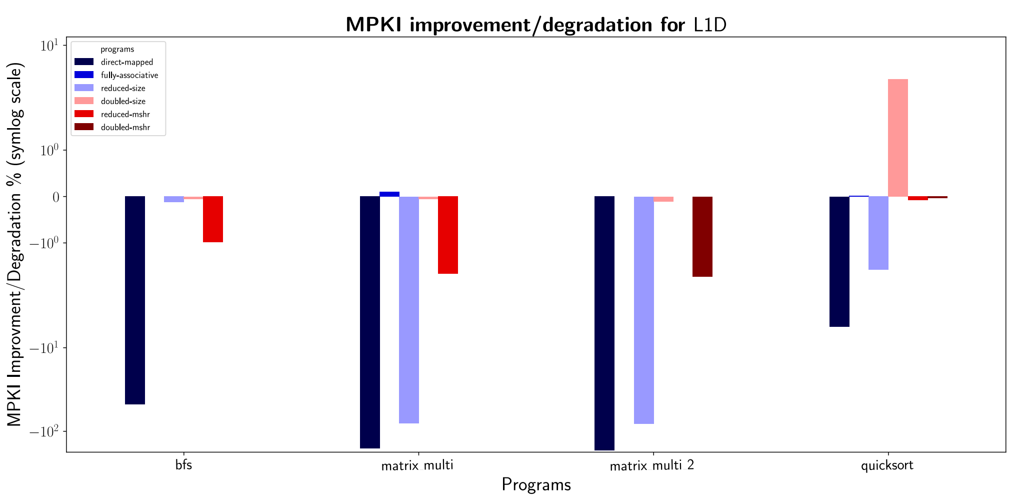
- We see an overall decrease in average miss latency for matrix_multi, matrix_multi_2 and quicksort in the L1D, L2C, LLC caches.

3.8 Summarizing Plots:

Attained IPC speedup normalised with respect to baseline (i.e) we consider baseline to be 1.0.



MPKI improvement/degradation plots (in `symlog` scale) with respect to the baseline for each of the caches: L1D, L1I, L2C, LLC. **A positive % means decrease in MPKI and vice versa.**



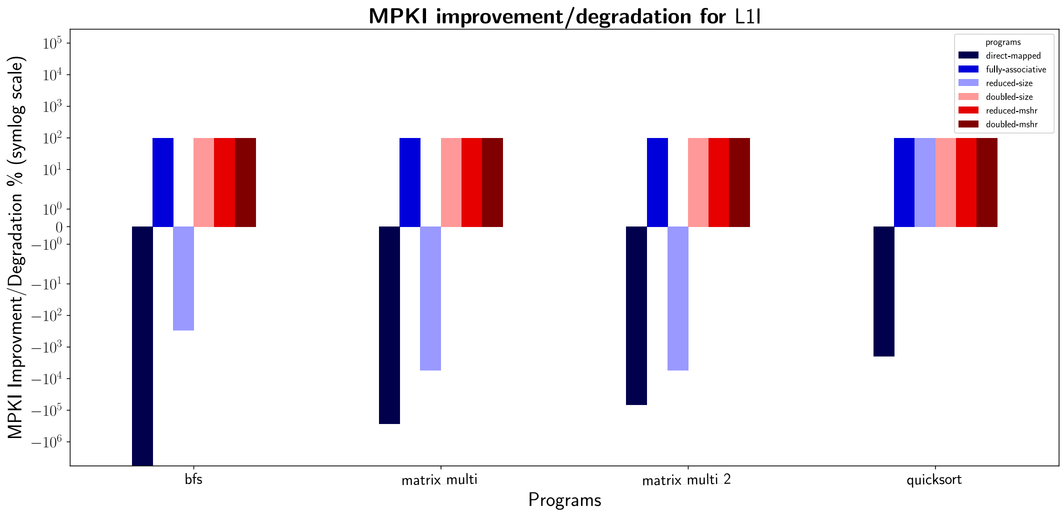


Figure 3.1: Affect of the 6 different settings on L1I cache. The graph looks unpleasant since L1I had ~ 0 MPKI in the baseline.

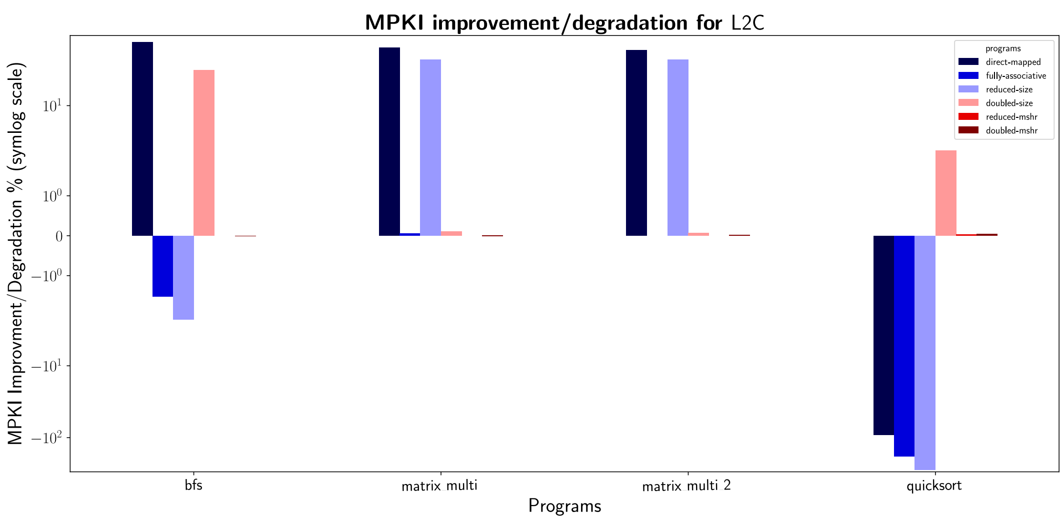


Figure 3.2: Affect of the 6 different settings on L2C cache with respect to baseline.

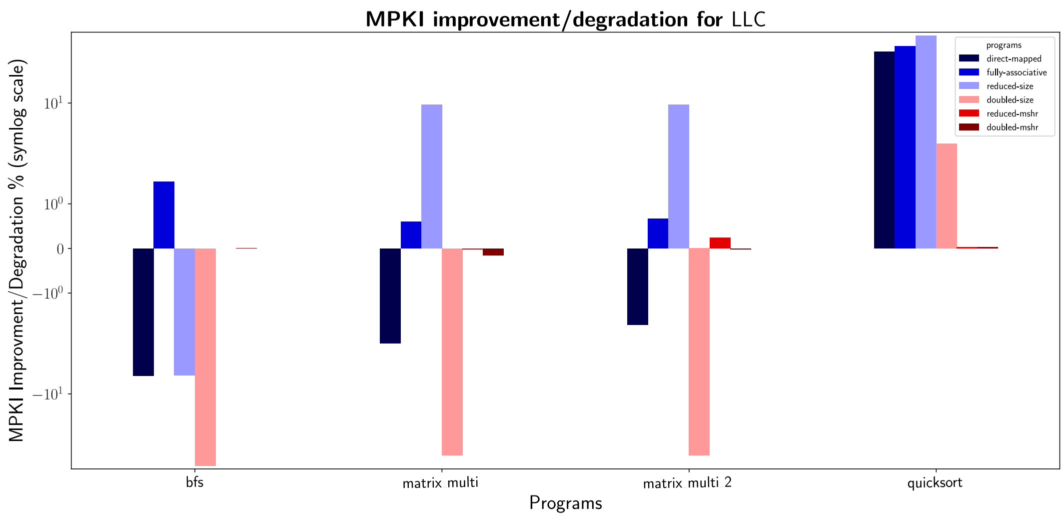


Figure 3.3: Affect of the 6 different settings on LLC cache with respect to baseline.

4. Contributions

Table 4.1: Contributions of each team member

Member	**Total Contribution**
Me	100%