Question 1)

a)

Correlation of different versions of program with the execution time and overall different memory hierarchy performance.

We have 6 different versions of the current program with tile size fixed as 32. Using perf to analyze the performance counters we observed following.

The mapping for versions throughout the problem is as follows.

Version 1 (V1): ijk

Version 2 (V2): jik

Version 3 (V3): jki

Version 4 (V4): ikj

Version 5 (V5): kij

Version 6 (V6): kji

For analyzing the part of program that is after initialization is done, we have used perforator with using perf in Linux.

Perforator uses ptrace to take performance counters of only certain required part of program for example here code has two functions initialization and multiplication, and we want performance counters for only multiplication function. Perforator places some interrupt instruction at the beginning of the profiled function which allows it to profile required function. At that point, Perforator will place the original code back (whatever was initially overwritten by the interrupt byte), determine the return address by reading the top of the stack, and place an interrupt byte at that address. afterwards it will start taking notes of performance counters. When the next interrupt occurs, it will have reached the return address and perf will stop noting performance counters from that part of program.

These programs were run on Linux OS with 16 GB DRAM, 12 Core machines.

The other machine specifications:

DRAM:	16 GB
L1 Data cache:	48 KB
L1 Instruction cache:	32 KB
L2 unified cache:	3 MB
L3 unified cache:	12 MB
Page size:	4096 Bytes
CPU clock speed:	in ideal state 2700 MHz
	In minimum state 800 MHz
Architecture:	X86_64

For matrix size: 2048

version	time-elapsed	page_fault	branch_ration	TLB-misses	L1-data-cache-misses-ration	L1-i-cache-misses-ration	LL-read-misses-ration	LL-write-misses-ration
v1	34.907143187s	0	2.943063	0.004913	6.111721	0.000022	40.044840	26.893135
v5	34.812262435s	0	2.942664	0.000527	6.116266	0.000027	9.301606	46.853147
v4	34.730156232s	0	2.943277	0.000429	6.113681	0.000023	9.396541	64.679487
v2	34.903632285s	0	2.942418	0.004767	6.115865	0.000026	31.205960	36.963599
v3	35.752298942s	0	2.943918	0.011866	6.115453	0.000030	65.260230	65.909091
v6	35.762504801s	0	2.943145	0.012251	6.110263	0.000034	65.013065	37.909135

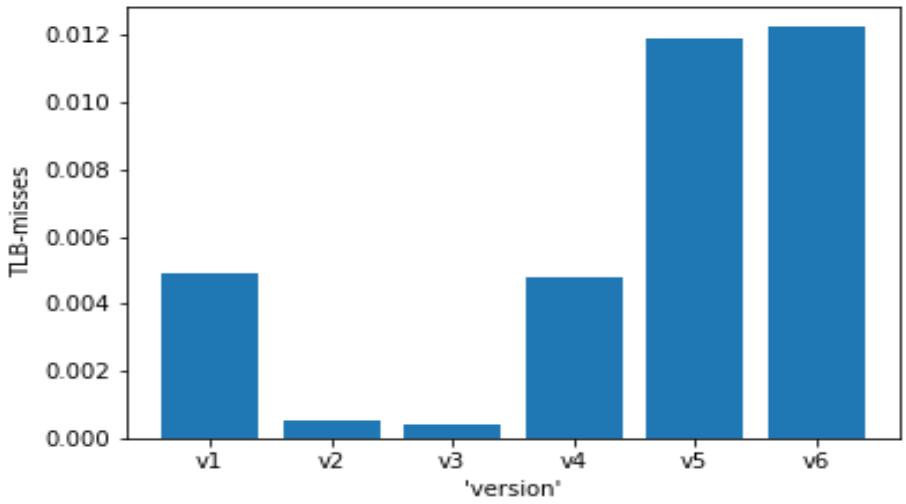
1. Due to un-uniformity of #Memory access in different versions of program it's better to analyze the ratio of misses at various levels. The graph of TLB misses ratio that is,

Tlb-miss ratio = #TLB misses / #Total TLB accesses

Vs. different versions of program is given below.

Conclusion: mostly versions from 1 to 4 are having same TLB-miss ratio, but version 3 takes the minimum number of misses per total access that has been done in that version.

Because #TLB accesses are bigger than TLB misses the ratio is coming out to be exceedingly small on scale.



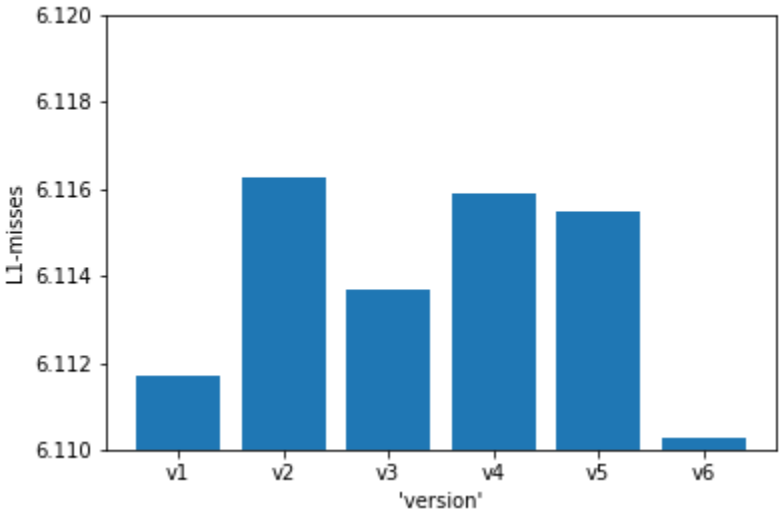
2. The graph of L1 cache misses ratio that is,

#L1 D-Cache misses / #Total L1 D-cache access

Vs. different versions of program is given below.

Conclusion: version 6 takes the minimum number of misses per total access that has been made in that version.

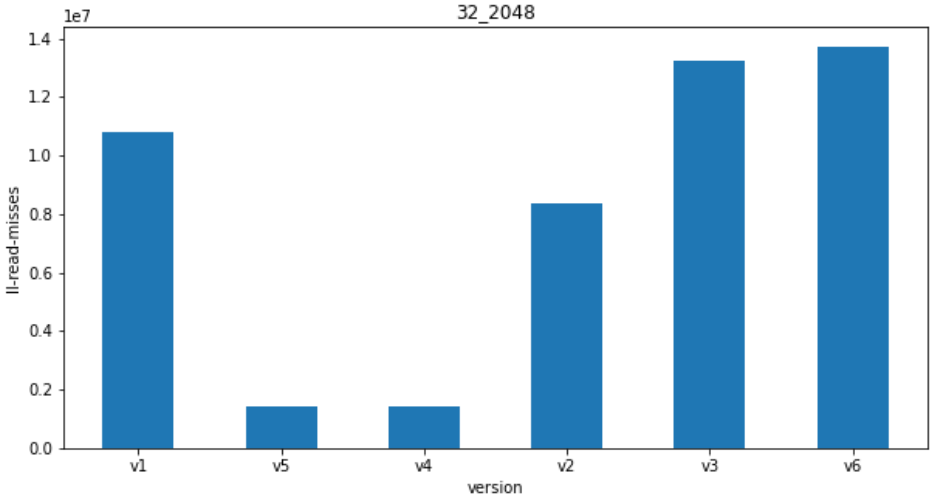
Here in the system that we run program, the cache associativity is 12, with each line size as 64 bytes. We have used double as datatype in program, which takes 8 bytes per matrix element. So, when there is cache miss, one cache line is fetched, means in total 8 matrix elements are fetched.



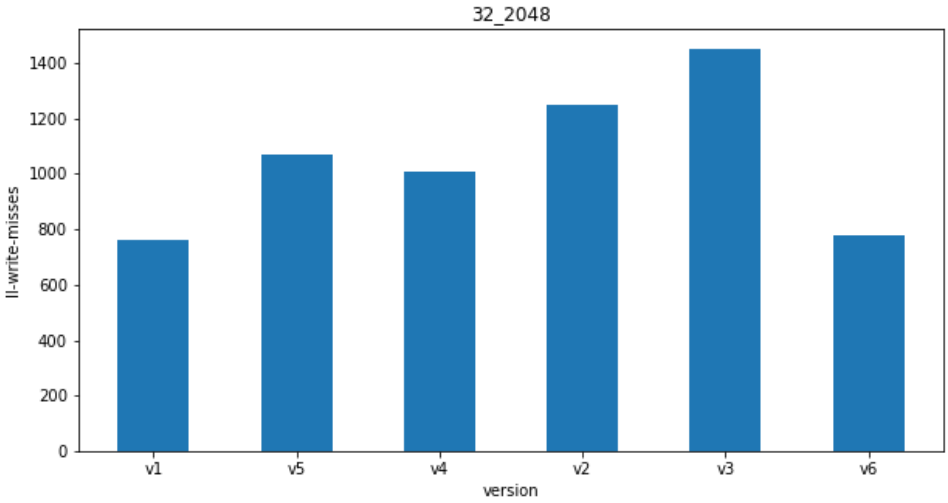
3. The graph of LL cache misses that is,

3.1 #LL read cache misses vs versions is given below.

The associativity of LL cache is 16, And Line size is 64 bytes.



3.2. The graph of #LL write cache misses vs versions is given below.

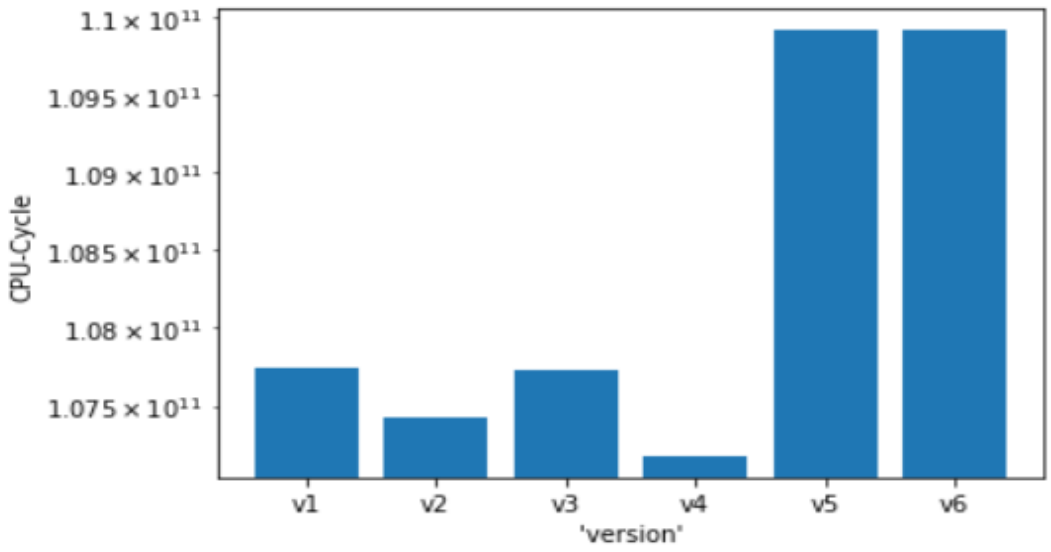


Conclusion: For last level memory hierarchy cache, version 4 gives least #read misses. As the last level cache is shared among all cores, it may be affected by other cores as well. In version 4 we can exploit good amount of spatial locality because of that we got minimum misses in version 4.

4. The graph of CPU-cycles taken by program vs different versions of program is given below.

Conclusion: version 4 takes minimum CPU-cycles to execute(approximately 34.7301 seconds).

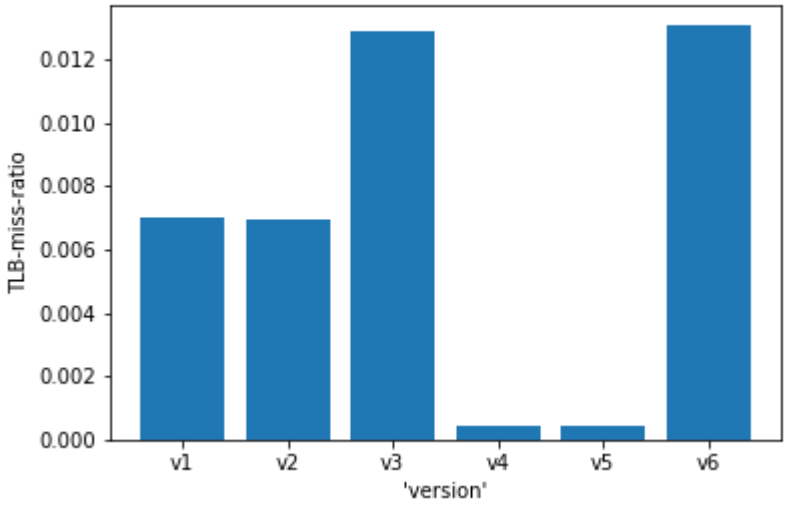
The way we are accessing elements in version 4, that is because both the input matrix are accessed raw wise we are exploiting maximum spatial locality and because of that #misses are lesser that results in less CPU-cycles to execute.


1.1 × 10^11
1.095 × 10^11
1.09 × 10^11
1.085 × 10^11
1.08 × 10^11
1.075 × 10^11
CPU-Cycle
v1
v2
v3
v4
v5
v6
'version'

For matrix size: 8192

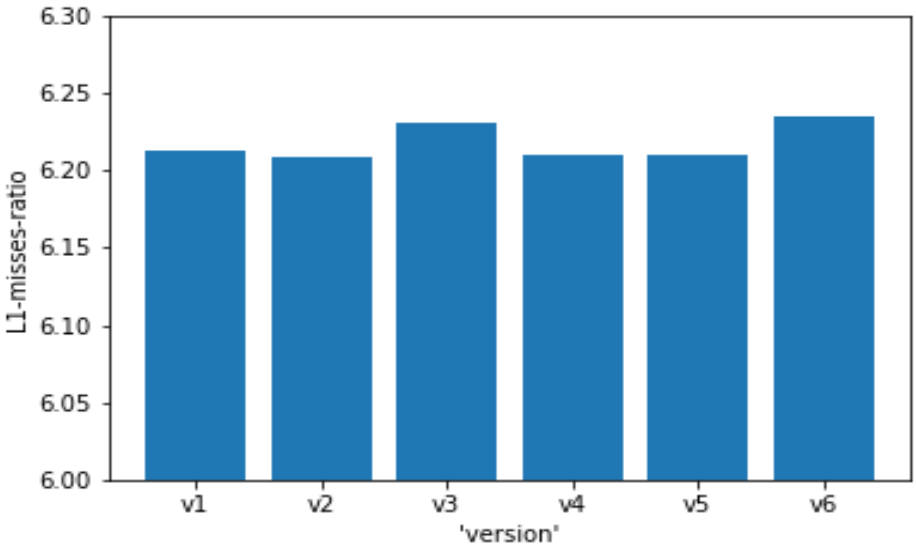
version	page_fault	branch-misses-ratio	TLB-misses	L1-data-cache-misses-ratio	L1-i-cache-misses-ration	LL-read-misses-ration	LL-write-misses-ration
v1	0	2.941038	0.006999	6.211954	0.000016	0.303617	0.172383
v2	0	2.940835	0.006970	6.208212	0.000015	0.292604	0.269939
v3	0	2.940831	0.012922	6.230500	0.000018	0.457285	0.026232
v4	0	2.940946	0.000454	6.209441	0.000015	0.030384	0.093082
v5	0	2.941109	0.000453	6.209472	0.000013	0.028703	0.085178
v6	0	2.941172	0.013086	6.234720	0.000018	0.475913	0.017185

1. The graph of TLB misses ratio Vs. different versions of program is given below.



Conclusion: version 4 and 5 takes minimum number of misses per total access that has been done in that version.

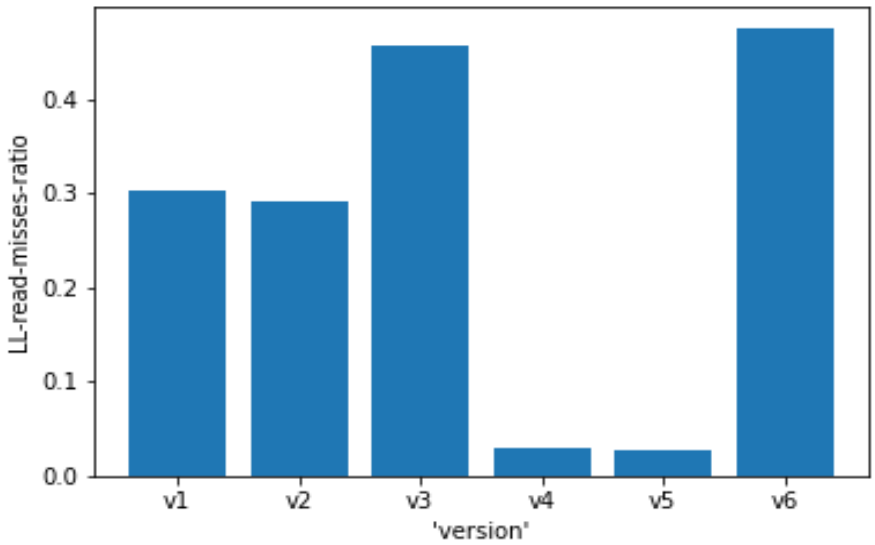
2. The graph of L1 cache misses ratio Vs. different versions of program is given below.



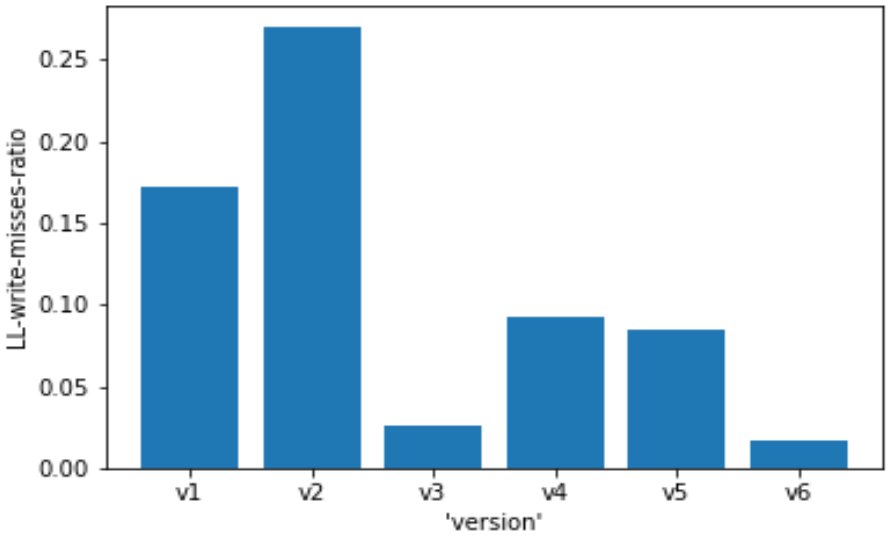
Conclusion: Almost all versions take the same number of misses per total access that has been made in that version.

3. The graph of LL cache misses that is,

3.1 #LL read cache misses vs versions is given below.

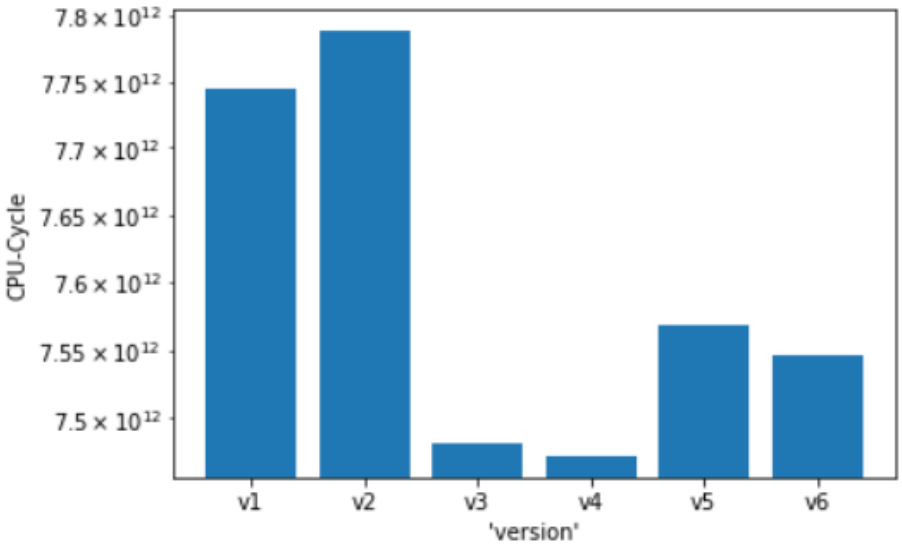


3.2 #LL write cache misses vs versions is given below.



Conclusion: version 4 and version 5 takes minimum number of LL read cache misses.

4. The graph of CPU-cycle that is taken by program vs different versions of program is given below.



Conclusion: version 4 takes minimum #of CPU-cycles.

Summary:

Input matrix elements are stored in raw major order in memory. In version 4 we are accessing both the input matrix raw wise, that exploiting the best possible spatial locality. So, memory performance is good, implying overall lower memory miss latencies. So, CPU cycles are reduced compared to other versions. Same way with matrix size as 2048 versions 5 and 6 are giving worst performance and for matrix size as 8192 versions 1 and 2 are giving worst performance.

b)

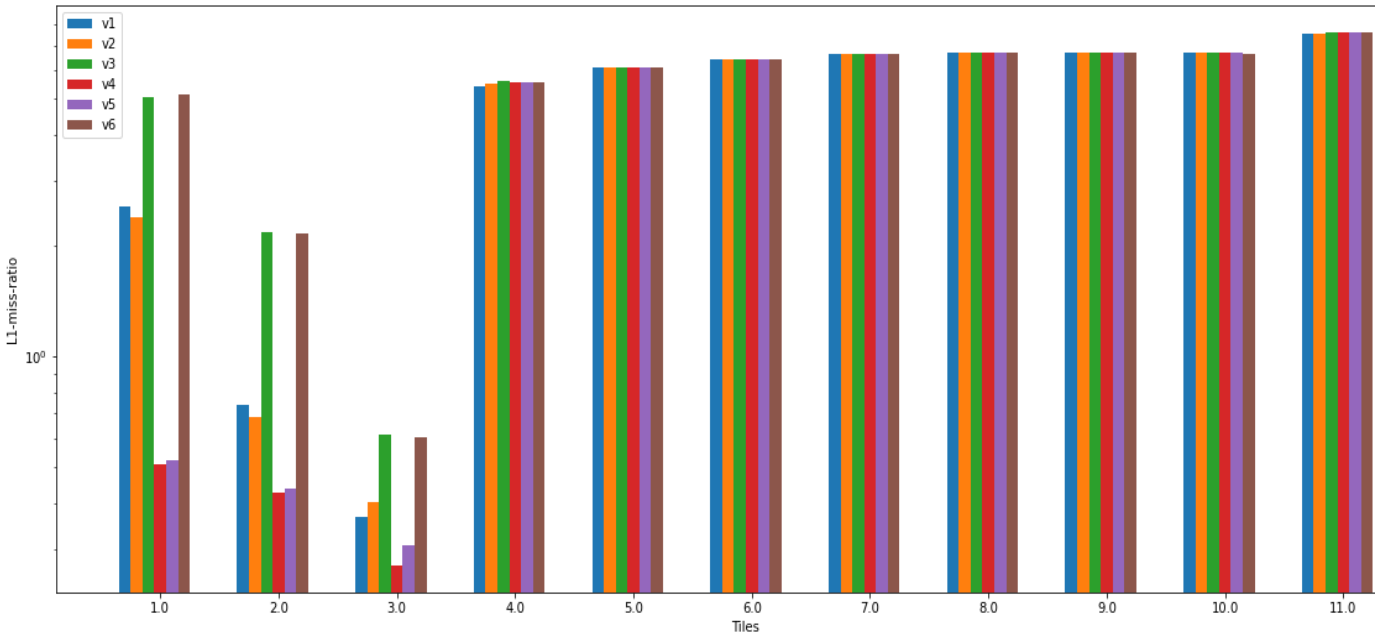
Exploration of different tile sizes:

Here we run 6 different versions of the program that we have for performance analysis for different tile sizes.

For multiplying matrix of the size 2048 we took different tile sizes as on range from different powers of 2 that is from 2^1 to 2^{11} . Here our tile sizes vary from 2 to 2048 i.e., final size of matrix itself. We also wanted to check simple matrix element by element wise multiplication performance with the other tile sizes, so we kept size as from 2 to 2048 also.

1. The graph below shows tile sizes verses L1- cache miss ratio when executed program for different 1 to 6 versions of programs by processor.

version	2	4	8	16	32	64	128	256	512	1024	2048
v1	2.565	0.738	0.369	5.433	6.111	6.433	6.635	6.662	6.675	6.694	7.506
v2	2.383	0.686	0.402	5.504	6.115	6.43	6.637	6.659	6.676	6.687	7.528
v3	5.076	2.175	0.611	5.594	6.115	6.432	6.638	6.66	6.679	6.701	7.557
v4	0.511	0.427	0.27	5.541	6.113	6.433	6.636	6.658	6.674	6.674	7.593
v5	0.521	0.437	0.307	5.554	6.116	6.43	6.637	6.66	6.675	6.695	7.566
v6	5.132	2.163	0.605	5.571	6.11	6.432	6.641	6.661	6.675	6.653	7.59

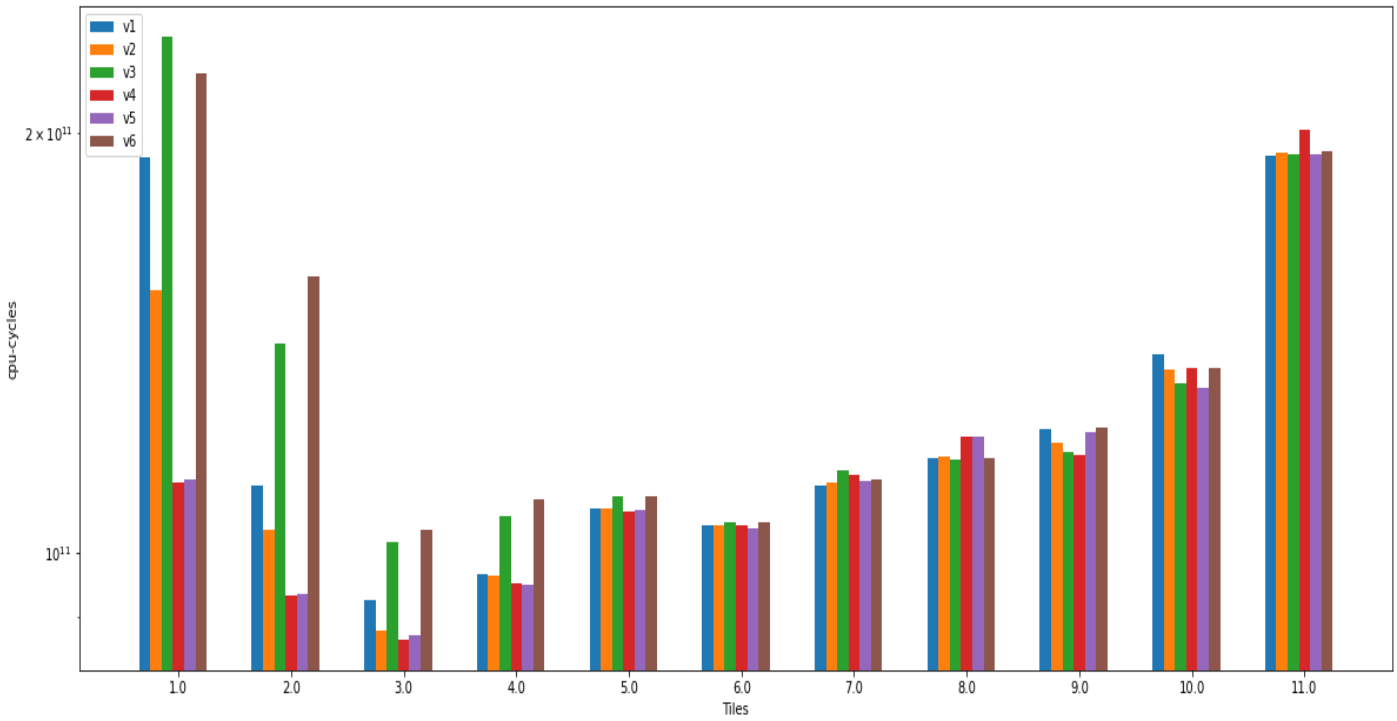


Conclusion: we observed that with tile size as 8 we obtained best L1 cache miss ratio performance vs different tile sizes. Then slight increase in tile size from 8 to next 16 increases L1 cache misses drastically and then that remains same way.

Because in the system that we run our program, cache line sizes for L1, L2, LL were 64 Bytes. And as we have used double data type to store elements in matrix, each element is of size 8 bytes. So, at max we can store 8 elements in one cache line. So, when one miss happened the entire block was fetched into cache and then we access addresses next, that all 8 next consecutive accesses were hit and then we took next tile. then per tile access only 1 cache miss happened.

Say we increase tile size from 8 to next powers of 2, then as cache line size is same, for one tile we need to fetch multiple lines from next level of memory, which eventually increases cache miss ratio overall. Because we need multiple cache lines to bring-in before changing tile, there is possibility that some cache line can be evicted from cache, which will increase the number of cache misses.

2. The graph below shows tile sizes verses CPU-cycles taken to execute program for different 1 to 6 versions of programs by processor.



Conclusion: tile size as $2^3 = 8$ gives the best performance with the least number of CPU-cycles needed.

Now for some versions CPU requires more cycles and for some versions it took very less cycles because the data that is each consecutive element in matrix might be stored in row major order in our organization.

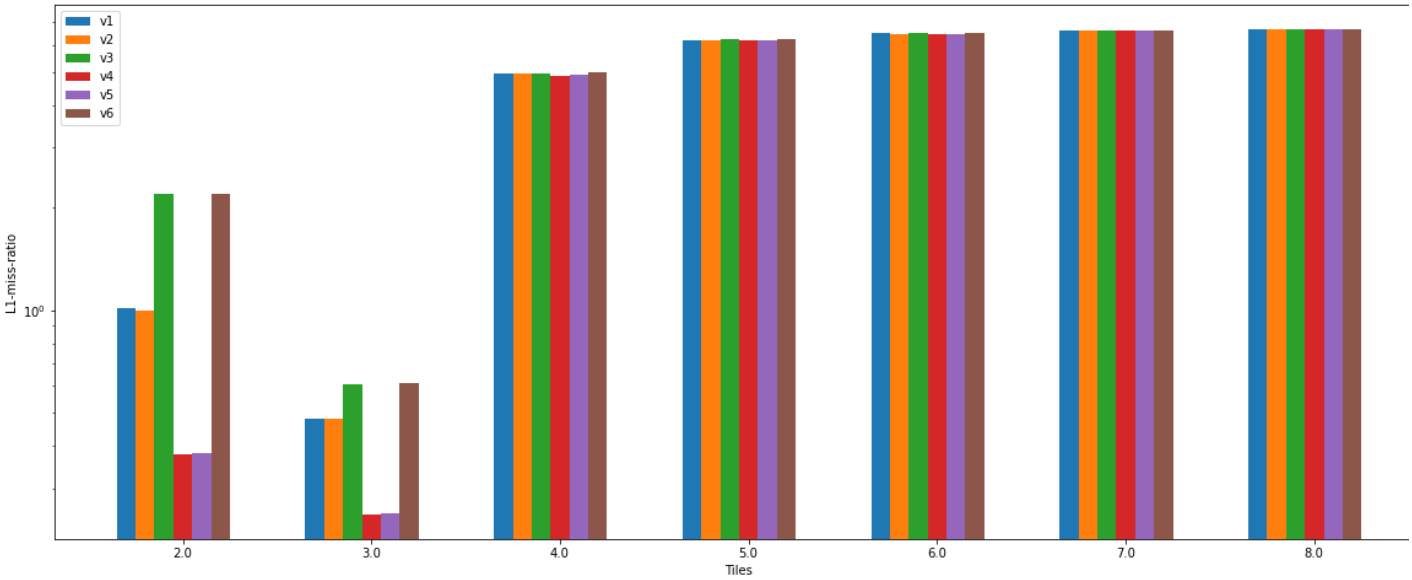
If we take tile size as 8 then, each tile is having raw of 8 elements and 8 such raw. The cache line also has 8 elements. That is why each consecutive access does not become cache miss and we have lower miss latency. Which leads to best memory performance and hence overall less CPU-cycles.

For matrix size: 8192

For multiplying matrix of the size 8192 we took different tile sizes as on range from different powers of 2.

3. The graph below shows tile sizes vs. L1 cache miss ratio when executed for different 1 to 6 versions of programs.

version	4	8	16	32	64	128	256
v1	1.01674	0.48122	4.94906	6.21195	6.49621	6.62949	6.683
v2	0.99728	0.48246	4.93341	6.20821	6.468459	6.624	6.6799
v3	2.20006	0.60874	4.95838	6.2305	6.5003	6.63044	6.68117
v4	0.37676	0.25128	4.88313	6.2094	6.48267	6.6396	6.6821
v5	0.38224	0.25294	4.89305	6.20947	6.4661	6.6294	6.6817
v6	2.20509	0.61059	4.9982	6.23471	6.49708	6.64906	6.68258

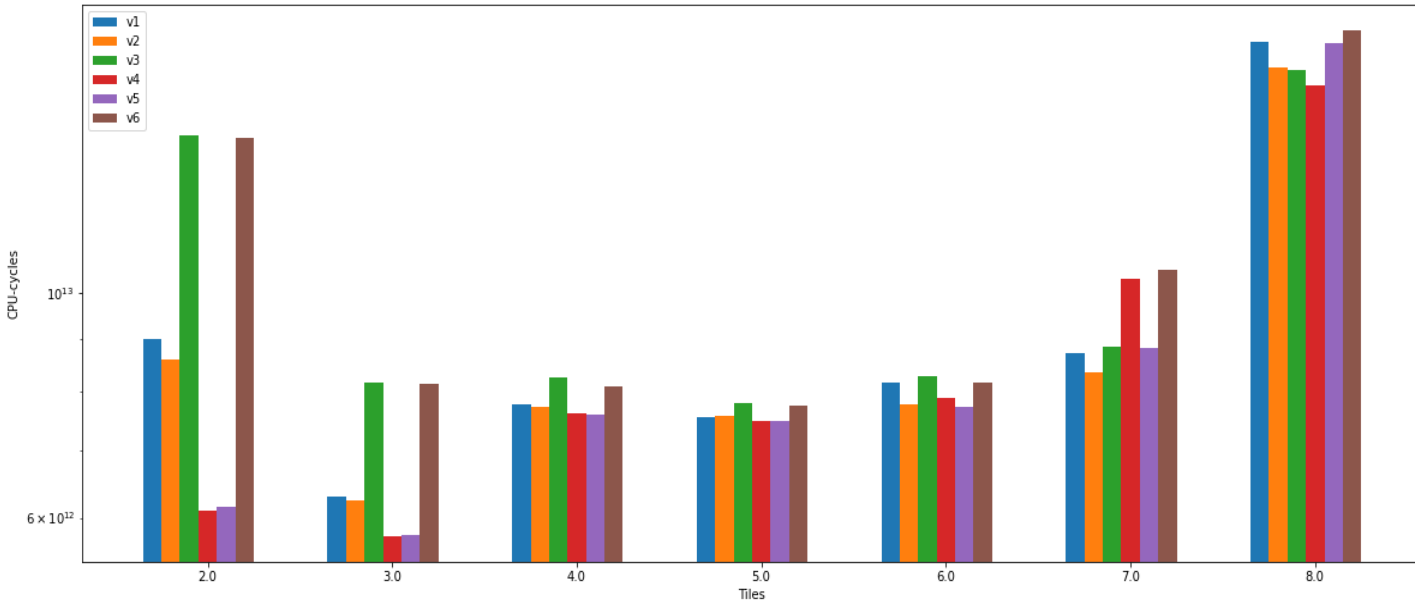


Conclusion: we observed that again with tile size as 8 we obtained best L1 cache miss ratio performance vs different tile sizes. then increase in tile size leads to more L1 cache misses.

4. The graph below shows tile sizes vs. CPU-cycles when executed for different 1 to 6 versions of programs.

version	4	8	16	32	64	128	256
v1	9.01091 E+12	6.29029 E+12	7.76039 E+12	7.54587 E+12	8.15954 E+12	8.71959 E+12	1.76468 E+13
v2	8.59298 E+12	6.23554 E+12	7.7151E +12	7.56894 E+12	7.75514 E+12	8.35458 E+12	1.66746 E+13
v3	1.42727 E+13	8.16111 E+12	8.25908 E+12	7.78779 E+12	8.27831 E+12	8.85649 E+12	1.65867 E+13

v4	6.1054E+12	5.75005 E+12	7.60441 E+12	7.48167 E+12	7.86768 E+12	1.03129 E+13	1.60156 E+13
v5	6.14347 E+12	5.77344 E+12	7.58138 E+12	7.47238 E+12	7.71528 E+12	8.8231E +12	1.76362 E+13
v6	1.42191 E+13	8.12317 E+12	8.0756E +12	7.744E+ 12	8.15974 E+12	1.05318 E+13	1.81427 E+13



Conclusion: tile size as $2^3 = 8$ gives the best performance with the least number of CPU-cycles needed.

Question 2)

Evaluate the performance of the given three microbenchmarks and analyze the performance of different memory scheduling schemes and the addressing schemes. In particular your analysis should include comparisons of row-buffer hit rates, bank-level parallelism, average memory access time, bandwidth exploited, etc.

Generation of traces:

We have used pin tool to generate cache filtered DRAM memory traces. We have also used RDTSC command along with to generate traces with timestamp also.

For that we made changes in allcache.cpp file as follows,

1) Added function and rdtsc command to generate time stamps also along with traces
2) Made one output file and out stream buffer where we write addresses when it was miss in ul2 and gone to DRAM.
3) Used normalization factor

We have used a pin tool which itself has additional overhead on time, so normalization of time stamp is required.

For generating time stamps we have used the normalization factor. Once we noted the starting timestamp say start time, then noted time stamp when CPU goes to DRAM say end time then,

Time stamp = (end time – start time) / normalization factor

The normalization factor for different benchmarks were different. We run the program without pin tool and noted the time taken to execute program. Then we run program along with pin tool to generate traces and then noted the time taken to execute with pin tool. Then,

Normalization factor = time taken with pin tool to execute / time taken without pin tool to simply run program

Then finally we generated traces for all benchmarks by using appropriate normalized factor.

The traces thar we generated can be found here:

traces

We have to implement 5 different policies.

1) FR-FCFS
2) OPEN
3) CLOSE
4) OPEN-4
5) FCFS

Description of benchmark 1:

Here, for first raw of first bank we accessed first column. Then for second raw accessing first column and repeated this process till all the rows are accessed and then changed the bank. Then repeated this process 6 times.

Description of benchmark 2:

Here, the benchmark is generating randomly the memory accesses.

Description of benchmark 3:

Here, first we accessed the element of first row, then accessed next column of same column and then repeated process for all banks.

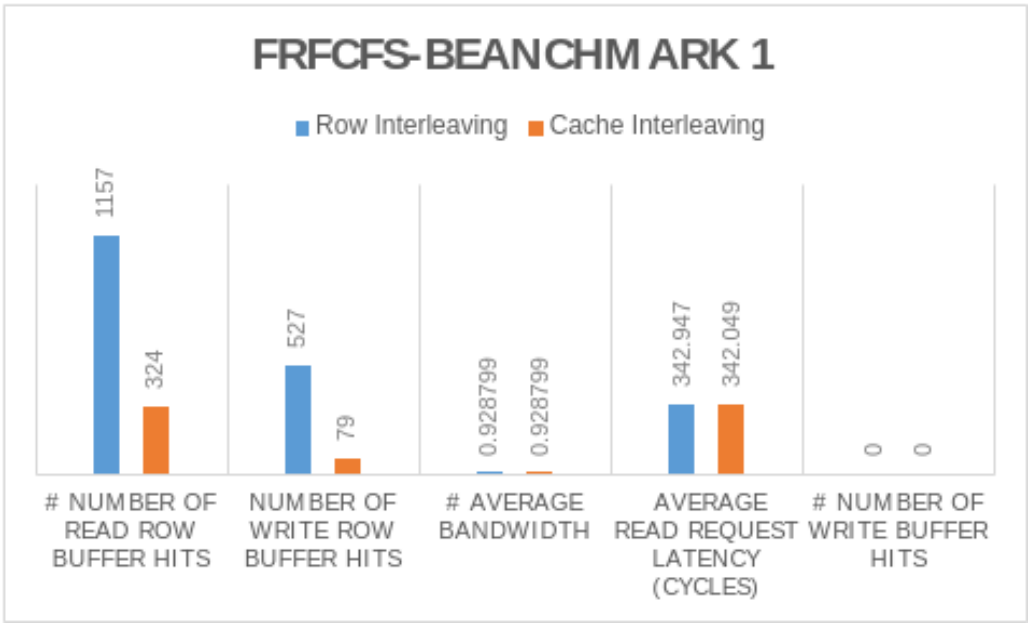
1) FR-FCFS:

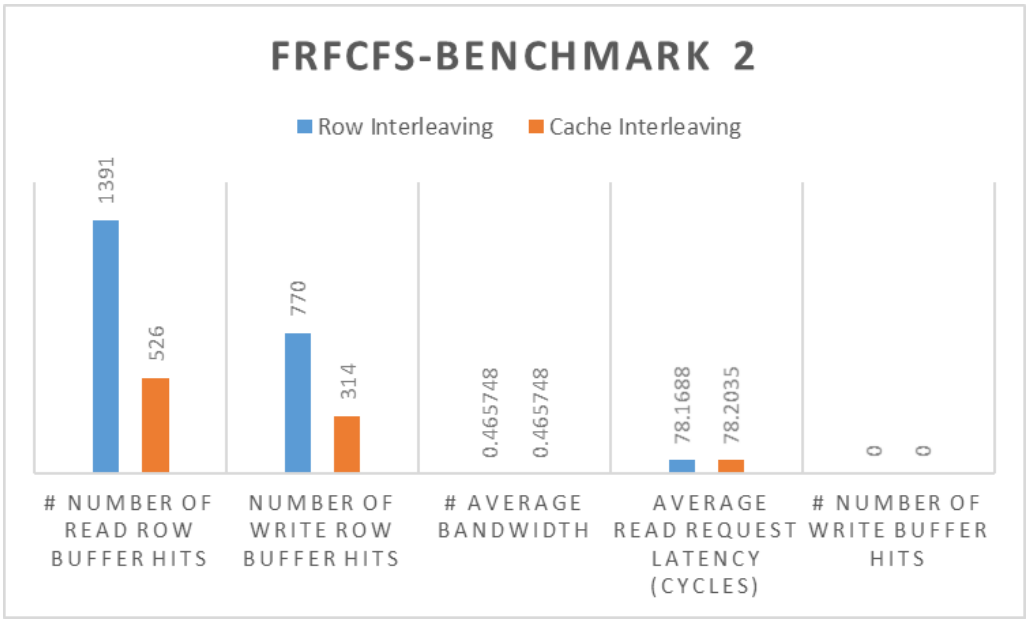
	B 1						
FR-FCFS			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency	# Number of write buffer hits
		Row Interleaving	1157	527	0.928799	342.947	0

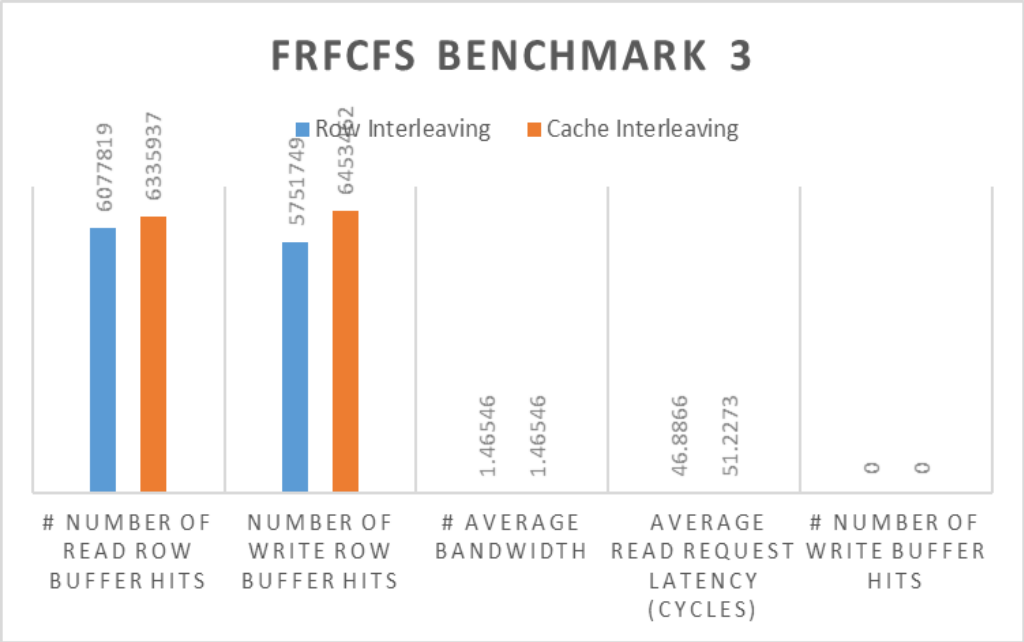
		Cache Interleaving	324	79	0.928799	342.049	0
	B 2						
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency	# Number of write buffer hits
		Row Interleaving	1391	770	0.465748	78.1688	0
		Cache Interleaving	526	314	0.465748	78.2035	0
	B 3						
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency	# Number of write buffer hits
		Row Interleaving	6077819	5751749	1.46546	46.8866	0
		Cache Interleaving	6335937	6453462	1.46546	51.2273	0

Here while implementing FR-FCFS, the average read latency of raw interleaving is lesser compared to cache interleaving. Also benchmark 3 is giving lowest average read latency because in benchmark 3 we have kind of structure that Because we are accessing each column of one row in bank and then pre-charging so there will be less number of row buffer misses.

The graphs of FR-FCFS along with different benchmarks is shown below.


FRFCFS-BEANCHM ARK 1
Row Interleaving
Cache Interleaving
1157
324
527
79
0.928799
0.928799
342.947
342.049
0
0
NUMBER OF READ ROW BUFFER HITS
NUMBER OF WRITE ROW BUFFER HITS
AVERAGE BANDWIDTH
AVERAGE READ REQUEST LATENCY (CYCLES)
NUMBER OF WRITE BUFFER HITS


FRFCFS-BENCHMARK 2
Row Interleaving
Cache Interleaving
1391
526
770
314
0.465748
0.465748
78.1688
78.2035
0
0
NUMBER OF READ ROW BUFFER HITS
NUMBER OF WRITE ROW BUFFER HITS
AVERAGE BANDWIDTH
AVERAGE READ REQUEST LATENCY (CYCLES)
NUMBER OF WRITE BUFFER HITS



2) OPEN:

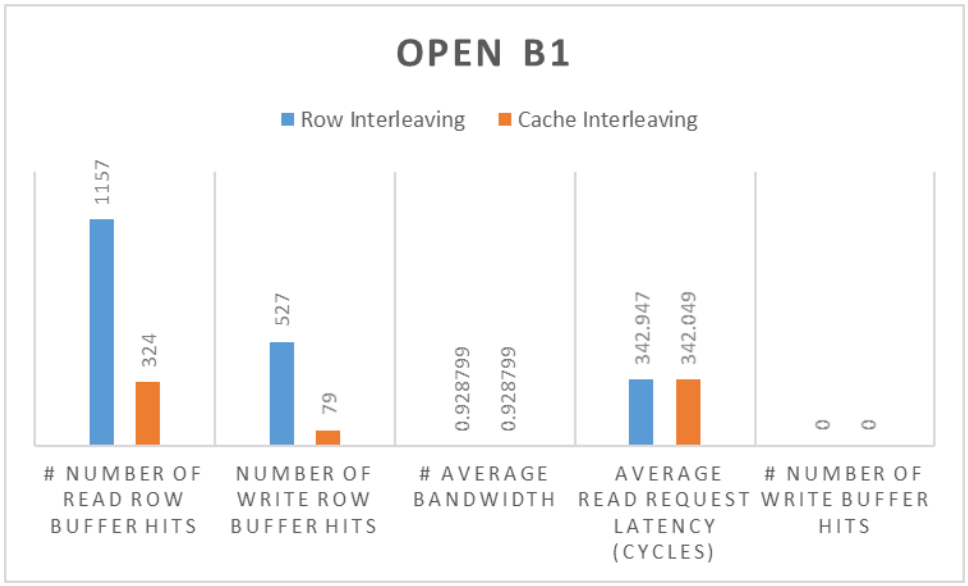
OPEN	B 1						
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency (cycles)	# Number of write buffer hits
		Row Interleaving	1157	527	0.928799	342.947	0
		Cache Interleaving	324	79	0.928799	342.049	0
	B 2						
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency (cycles)	# Number of write buffer hits
		Row Interleaving	1391	770	0.465748	78.1688	0
		Cache Interleaving	526	314	0.465748	78.2035	0
	B 3						

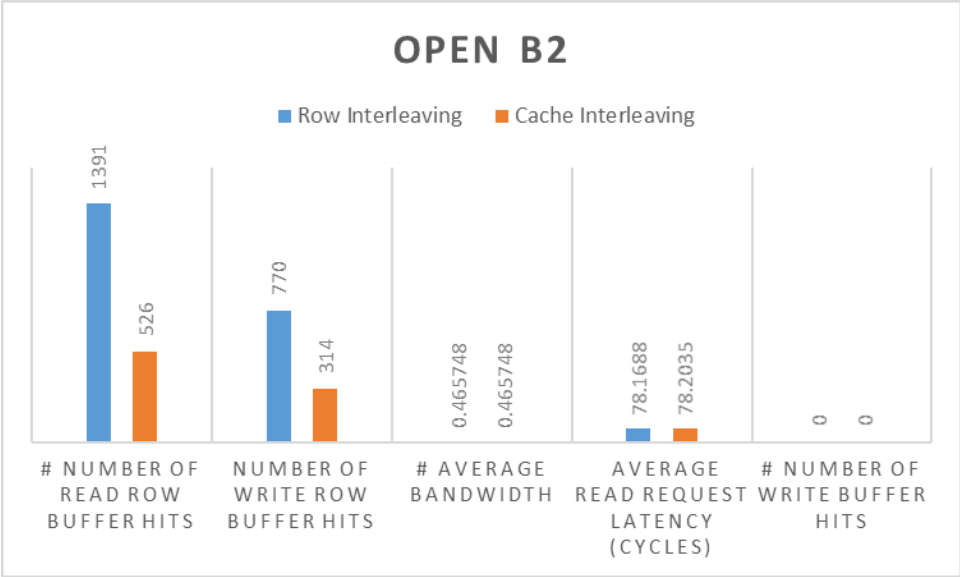
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency (cycles)	# Number of write buffer hits
		Row Interleaving	6077819	5751749	1.46546	46.8866	0
		Cache Interleaving	6335937	6453462	1.46546	51.2273	0

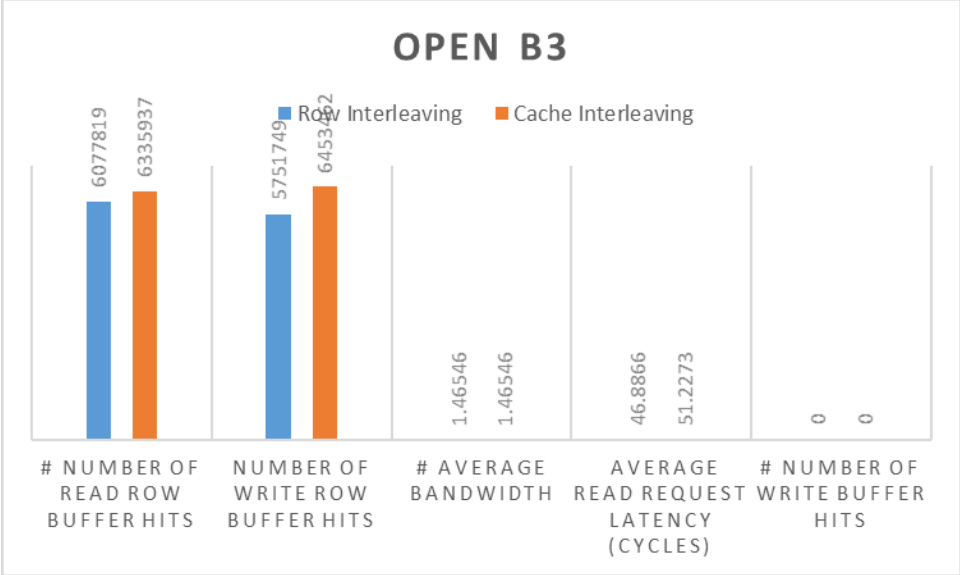
Open row buffer policy:

A bank is only pre-charged if there are pending references to other rows in the bank and there are no pending references to the active row. The open policy should be employed if there is significant row locality, making it likely that future references will target the same row as previous references did.

While using open row buffer policy, we observed that again Benchmark 3 is giving best performance possible and again row interleaving is good.







3) CLOSE:

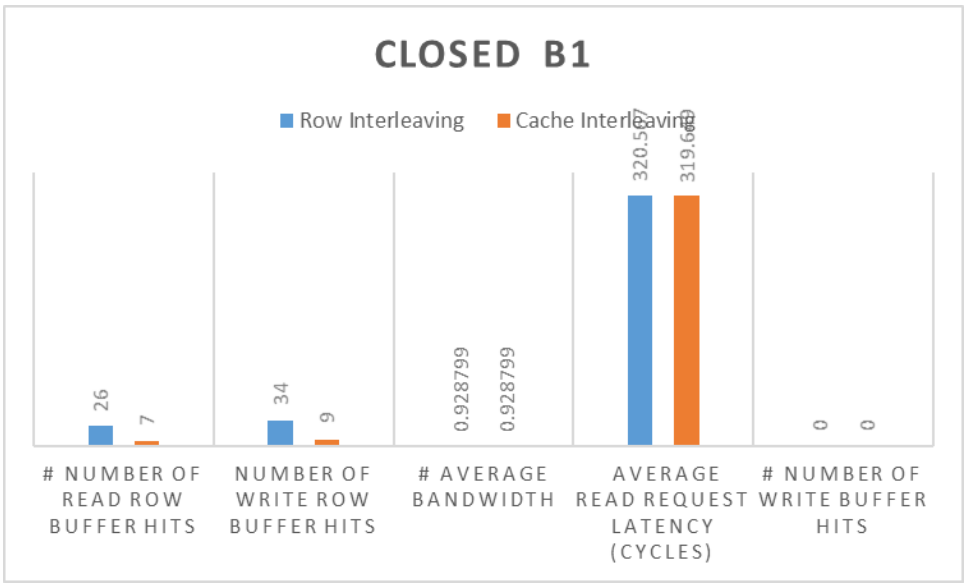
	B 1						
Closed			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency (cycles)	# Number of write buffer hits
		Row Interleaving	26	34	0.928799	320.567	0

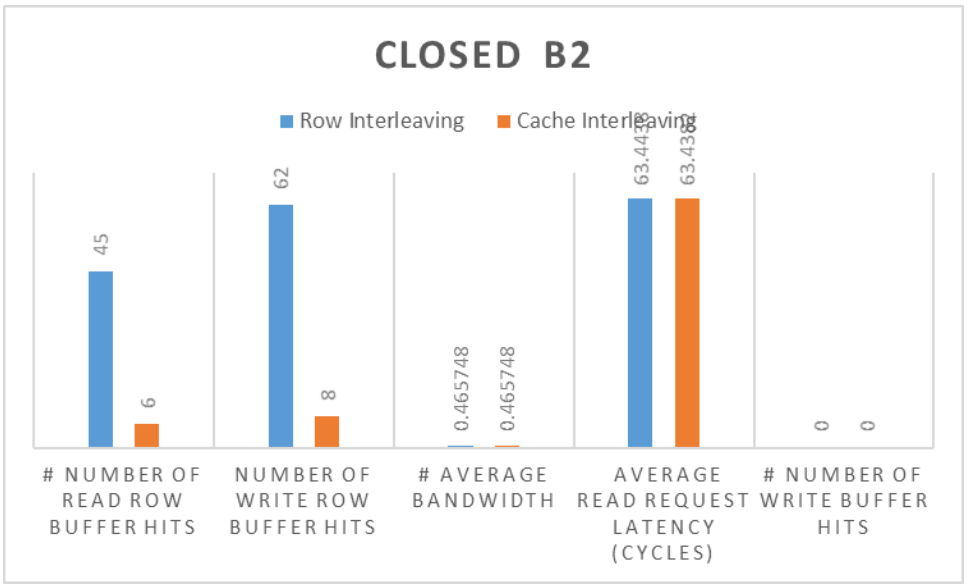
		Cache Interleaving	7	9	0.928799	319.649	0
	B 2						
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency (cycles)	# Number of write buffer hits
		Row Interleaving	45	62	0.465748	63.4438	0
		Cache Interleaving	6	8	0.465748	63.4382	0
	B 3						
			# Number of read row buffer hits	Number of write row buffer hits	# Average bandwidth	Average read request latency (cycles)	# Number of write buffer hits
		Row Interleaving	63	52	1.46546	66.2734	0
		Cache Interleaving	4	6	1.46546	294.255	0

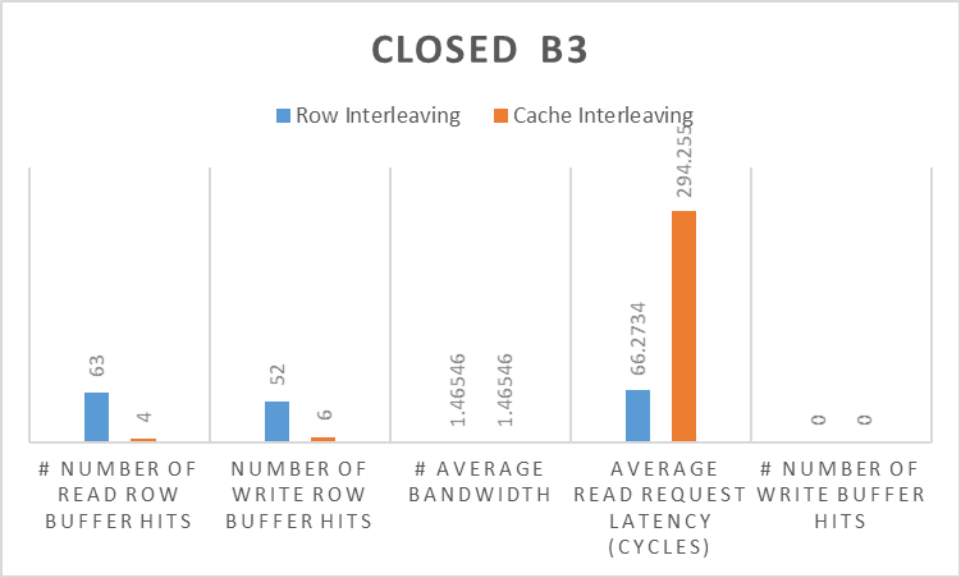
Closed row buffer policy:

A bank is precharged as soon as there are no more pending references to the active row. The closed policy should be employed if it is unlikely that future references will target the same row as the previous set of references.

For closed row buffer policy, benchmark 2 is giving minimum average read latency because addresses in banks are accessed randomly there we can exploit some kind of optimization. But in benchmark 1 and 3 after accessing column we are pre-charging it but there might be another consecutive column access which again leads to row buffer miss. Which eventually leads to more latency.


CLOSED B1
Row Interleaving
Cache Interleaving
26
7
34
9
0.928799
0.928799
320.527
319.667
0
0
NUMBER OF READ ROW BUFFER HITS
NUMBER OF WRITE ROW BUFFER HITS
AVERAGE BANDWIDTH
AVERAGE READ REQUEST LATENCY (CYCLES)
NUMBER OF WRITE BUFFER HITS


CLOSED B2
Row Interleaving
Cache Interleaving
45
6
62
8
0.465748
0.465748
63.4428
63.4384
0
0
NUMBER OF READ ROW BUFFER HITS
NUMBER OF WRITE ROW BUFFER HITS
AVERAGE BANDWIDTH
AVERAGE READ REQUEST LATENCY (CYCLES)
NUMBER OF WRITE BUFFER HITS



Summary:

For all of these 3 benchmarks, overall the FR-FCFS and Open are performing good compared to closed, because every time it is pre-charging. Among those policies row interleaving is performing good.