HIGH PERFORMANCE COMPUTER ARCHITECTURE

ASSIGNMENT 1

REPORT 1

Arpit Mishra (SR no.24131): arpitmishra@iisc.ac.in

Ayush Kumar Singh (SR no.24851): ayushs@iisc.ac.in

----------Ques 1(a)----------

We are required to use the standard O(n3) matrix multiplication algorithm. We have measured the performance of these different versions on two different matrix sizes of (2048 x 2048) and (8192 x 8192).

SYSTEM CONFIGURATION : We have performed the analysis on a system with following specifications.

- Intel(R) core™ i5-8300H cpu@2.30GHz, 2304MHz ,4 cores,8 logical processors
- RAM:8GB
- L1 Cache size=256KB
- L2 Cache size=1MB
- L3 Cache size=8 MB
- Page size=4KB

IMPLEMENTATION:
Each variant of the matrix multiplication was implemented in C, adhering to the following loop orders:

Standard Loop Orders:

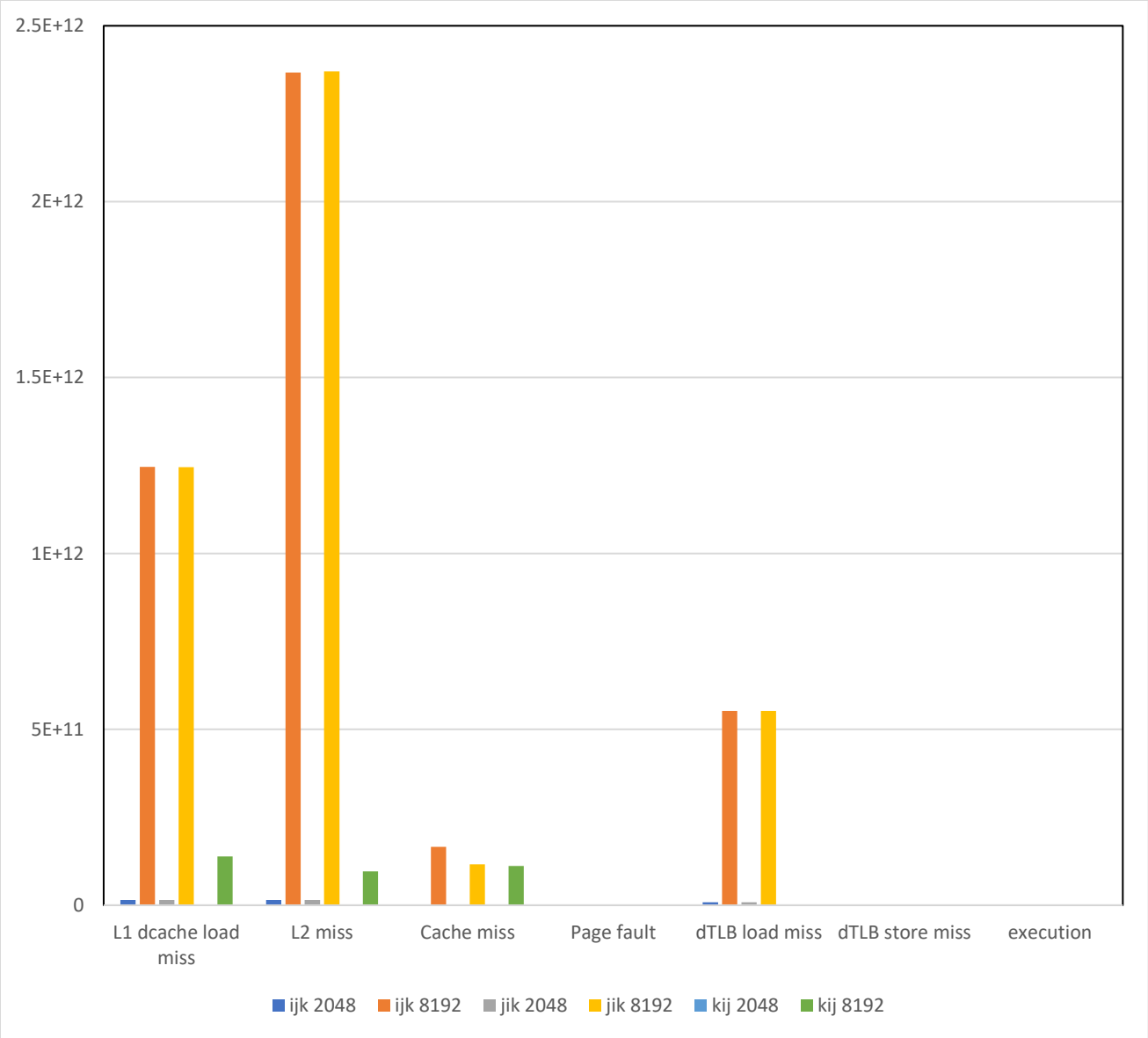
1. **(i, j, k)**: Iterates through rows and columns sequentially.
2. **(j, i, k)**: Iterates through columns first, then rows.
3. **(k, i, j)**: Iterates through layers (k) first, then rows and columns.

PERFORMANCE MEASUREMENTS: For performance measurement we have used PERF tool to capture various performance counters

- compile the program using : gcc mat_mul.c
- it will generate the a.out file
- check general stats : perf stat ./a.out
- we have measured the following performance counters : perf stat -e L1-dcache-loads,L1-dcache-load-misses,L1-icache-loads,L1-icache-load-misses,L2_rqsts.all_demand_data_rd,L2_rqsts.miss,dTLB-loads,dTLB-load-misses,dTLB-stores,dTLB-store-misses,iTLB-loads,iTLB-load-misses,page-faults,cache-misses ./a.out

RESULT

	Size	L1 dcache-load miss	L2 miss	Cache miss	Page Fault	dTLB load miss	dTLB store miss	execution
ijk	2048	15134077807	14673667994	1646127978	24668	8630742404	563398	85
ijk	8192	1246473109956	2366825825609	165897946777	393415	552502865398	34886201	6195
jjk	2048	15122547132	14966490455	1770588738	24667	8618386220	4194591	59
jik	8192	1245179546288	2370173216894	116780950246	393415	552481472284	97185260	4569
kij	2048	1289167970	1541065368	1525615353	24668	21087595	239456	33
kij	8192	138507812850	96722192729	111775353543	393414	1143702102	9204741	2248



ANALYSIS: Since each page is 4 KB (4096 bytes) and each matrix element is a double (8 bytes), each page can hold:

Elements per page=4096 /8 =512 element.

Each page can store 512 contiguous elements of a matrix in memory.

In Matrix multiplication

$$\boldsymbol{C[i][j] = \sum_{k=0}^{n-1} A[i][k] * B[k][j]}$$

Since arrays are stored in row-major order in the memory , the order in which we access the elements affects in terms of execution times, page faults, cache misses etc.

Considering the case of page faults,

Array A will be stored in the memory in the following manner(taking 2048*2048 dimension)

PAGE 1 ->	**A[0][0]**	**A[0][1]**	.	.	.	.	.	.	**A[0][511]**
PAGE 2->	**A[0][512]**	**A[0][513]**	.	.	.	.	.	.	**A[0][1023]**
.	.								.
.	.								.
.	.								.
.	.								.
.	.								.
PAGE 2^{13} ->	**A[2047][1536]**	.	.	.	.	.	.	.	**A[2047][2047]**

Array B and C will also have a similar representation.

- For matrix A, accessing a full row (A[i][k] for k = 0 to 2047) will involve 2048 / 512 = 4 page accesses per row, and there are 2048 rows, so there will be a total of 2048 * 4 = 8192 page accesses for matrix A.
- For matrix B, since we're accessing elements across different rows (i.e., B[k][j] for k = 0 to 2047), every new access to a different row can result in a page fault. Thus, for every j, we'll access 2048 rows of matrix B, resulting in 2048 / 512 = 4 page accesses per column. Since we perform this for every value of j (2048 columns), the total page accesses for B will be 2048 * 4 = 8192.
- For matrix C, each element C[i][j] is accessed once, so there will be one page access every 512 elements. Given that there are 2048 * 2048 = 4,194,304 elements, and each page holds 512 elements, there will be 4194304 / 512 = 8192 page accesses for matrix C.

Total page faults=3*8192 = 24000(approx)

OBSERVATION: The (k, i, j) loop order performs better in terms of execution time and cache misses, indicating better cache locality. This pattern suggests that accessing memory in a manner that aligns with how data is stored in memory (row-major order) improves performance.

----------Ques 1(b)----------

After enabling huge page support on the system, we map using the mmap.

void addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS |*

MAP_HUGETLB, -1, 0);

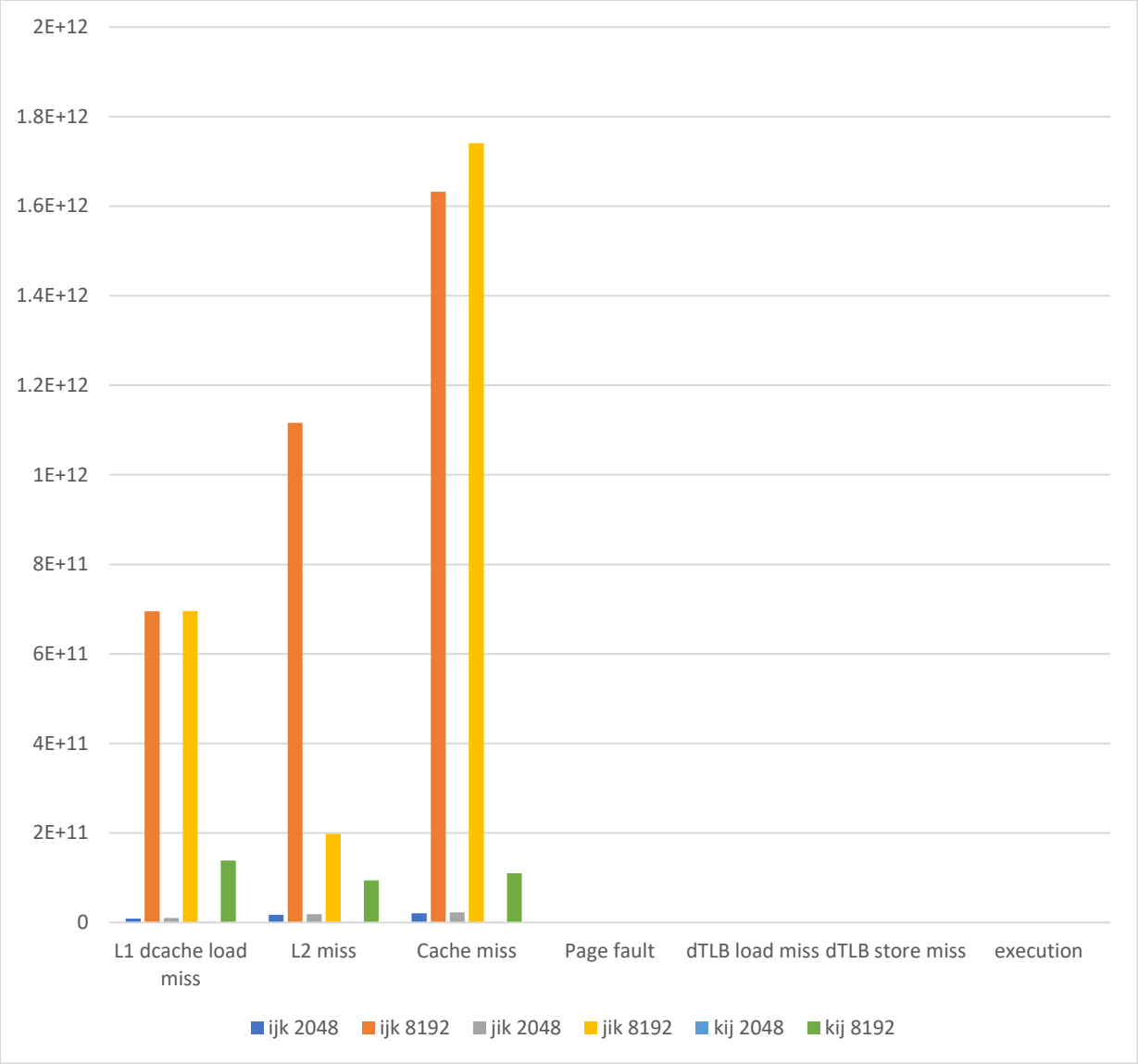
where size denotes the size of the mapping in bytes.

OBSERVATIONS:

- Using huge pages, we increase the reach of the TLB , and hence decrease the number of TLB misses
- Using huge pages, the number of page faults are also decreasing as on each page fault we are bringing larger data than in case of using normal pages.
- The execution time tends to reduce specially for larger size arrays.
- When using huge pages, the effective page size increases, meaning that a larger contiguous block of memory is assigned to each page. In matrix multiplication, this can lead to more cache lines mapping to the same cache set if the memory accesses are regular .
- As a result, memory access patterns in the matrix multiplication loop orders can cause memory blocks to map to the same cache sets. This increases the likelihood of conflict misses because multiple data items will be competing for the same cache lines.

RESULT

Order	Size	L1 dcache-load miss	L2 miss	Cache miss	Page Fault	dTLB load miss	dTLB store miss	execution
ijk	2048	9005619656	17240474737	20983396368	116	175438	313210	162
ijk	8192	695350916053	1116463413896	1632224583597	871	15058746	21028341	10534
jjk	2048	9977991607	18419712992	22723527496	112	210484	346560	169
jik	8192	695604511723	1197757810350	1740811424243	872	69041406	24700116	11163
kij	2048	1176545033	1517791579	1571315787	115	49339	83613	32
kij	8192	138219712304	94236164065	110304445900	871	3784774	5710946	2155

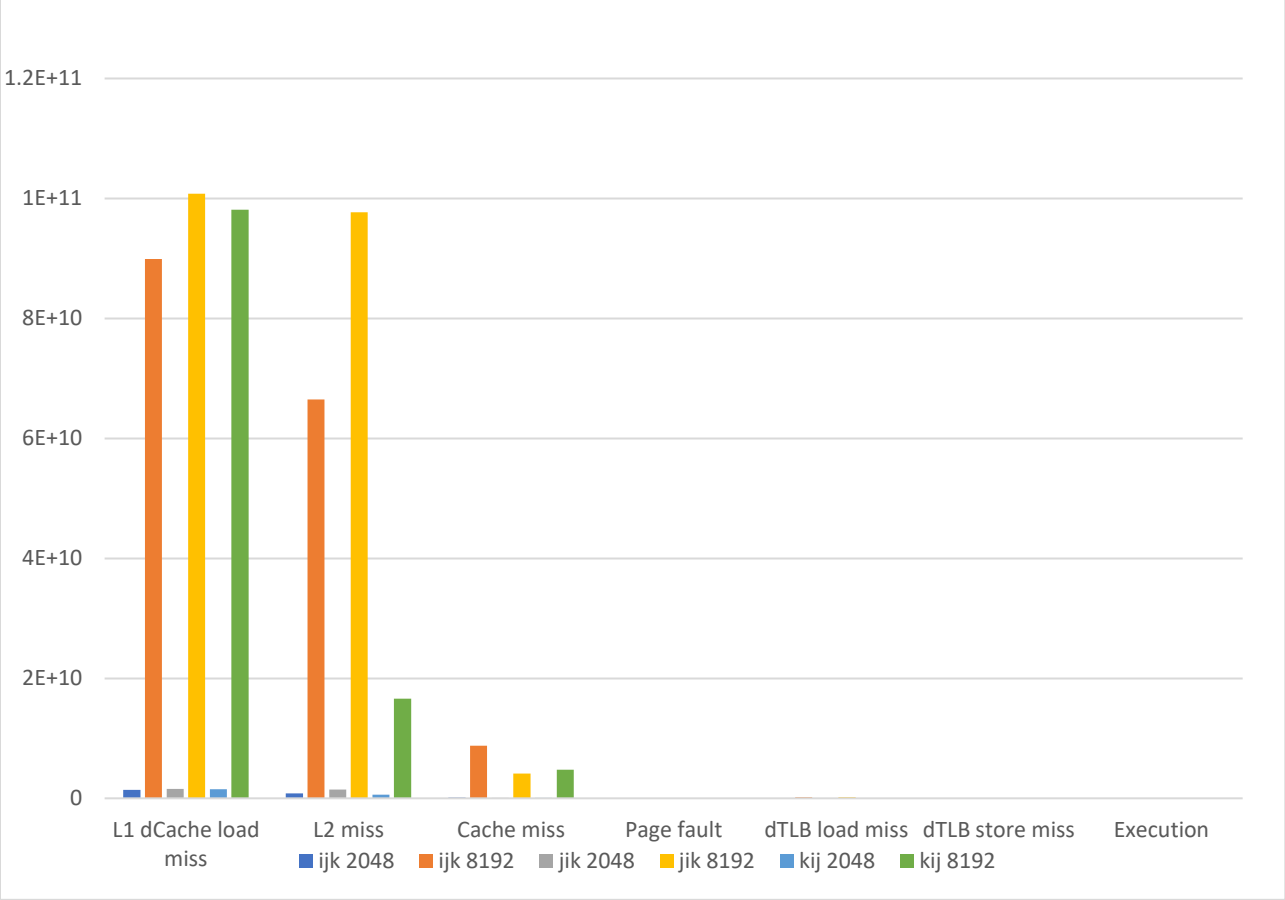


CONCLUSION: The kij loop order showed the best overall performance due to its memory access pattern, which efficiently utilizes the memory hierarchy. The use of huge pages significantly reduced page faults and TLB misses, improving execution time for large matrix sizes (8192x8192). The jik order, had the worst performance for larger matrices due to high cache misses, especially in the L1 and L2 levels, which are directly tied to its poor data locality.

----------Ques 1(c)----------

For each of the variants, we have created a corresponding blocked/tiled version with a tile size of 64. Tiled versions tend to have less cache misses than the untiled version of the same code. The tiled versions aims to improve cache utilization by processing small, contiguous blocks of data that fit into the cache.

Order	Size	L1 dcache-load miss	L2 miss	Cache miss	Page Fault	dTLB load miss	dTLB store miss	execution
ijk	2048	1431967930	843182986	124522771	24666	2839738	267442	42
ijk	8192	89921091084	66466774070	8749604553	393415	171517060	8253187	2551
jjk	2048	1595011319	1444050469	76643345	24667	3056098	233048	42
jik	8192	100786952773	97715910918	4147139838	393415	173162279	7524030	2381
kij	2048	1543587287	609560424	57251876	24666	587249	227646	36
kij	8192	98111873954	16619891018	4763282450	393415	38690326	7559346	2186



OBSERVATIONS:

- The tiled version has less number of cache misses than the untiled version of the same code.
- The tiled version has less number of TLB misses than the untiled version of the same code.
- Execution time is getting improved by better memory utilization
- kij loop order provides the best overall performance across both matrix sizes, likely due to its more favorable memory access pattern.

ANALYSIS:

Tiling ensures that when data is loaded into the cache, multiple operations are performed on it before moving to the next block. In standard matrix multiplication, the algorithm might load an element into the cache, use it once, and then discard it. With tiling, a block of data (e.g., 64x64 elements) is loaded into the cache, and multiple operations are performed on that data before it's evicted. This reduces cache misses, especially at the L1 and L2 levels.

CONCLUSION: Tiling matrix multiplication improves both execution time and memory performance by reducing cache and TLB misses. The results show that the kij loop order provides the best overall performance. The improvements due to tiling are more visible in the 8192*8192 matrix multiplication.

----------Ques 2(a)----------

We are required to evaluate different branch predictors using the Champsim simulator.

Traces used for evaluating the predictors are:

- Cassandra_phase1_core3
- Cloud9_phase2_core3
- Classification_phase2_core3

The following steps are needed to be performed:

1. Setup Champsim(clone it from github)
2. Enable the required predictor for which performance is to be measured
3. Champsim comes with the implementations of few predictors like bimodal, gshare, perceptron but for TAGE we need to integrate any open source code of TAGE with the Champsim.
4. Simulate the benchmark traces by
   bin/champsim --warmup-instructions 200000000 --simulation-instructions 500000000 ~/traces/cassandra_phase1_core3.trace.xz
5. The simulation is done after doing warm-up on 200 million instructions.

BIMODAL PREDICTOR

Trace	MPKI	IPC	Accuracy
Cassandra_phase1_core3.trace.xz	0.03471	0.5124	98.89
Cloud9_phase2_core3.trace.xz	0.6239	0.9475	96.47
Classification_phase2_core3.trace.xz	0.00066	0.7651	99.98

GSHARE PREDICTOR

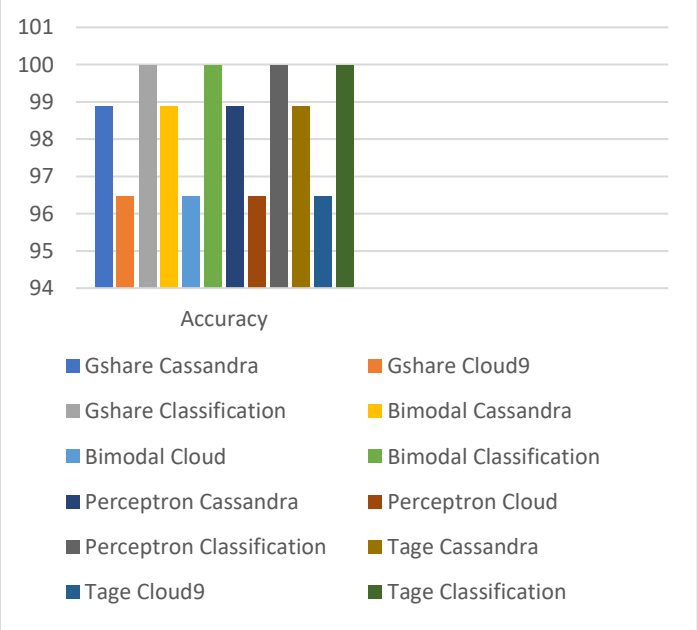
Trace	MPKI	IPC	Accuracy
Cassandra_phase1_core3.trace.xz	0.03471	0.5124	98.89
Cloud9_phase2_core3.trace.xz	0.6239	0.9475	96.47
Classification_phase2_core3.trace.xz	0.00066	0.7651	99.98

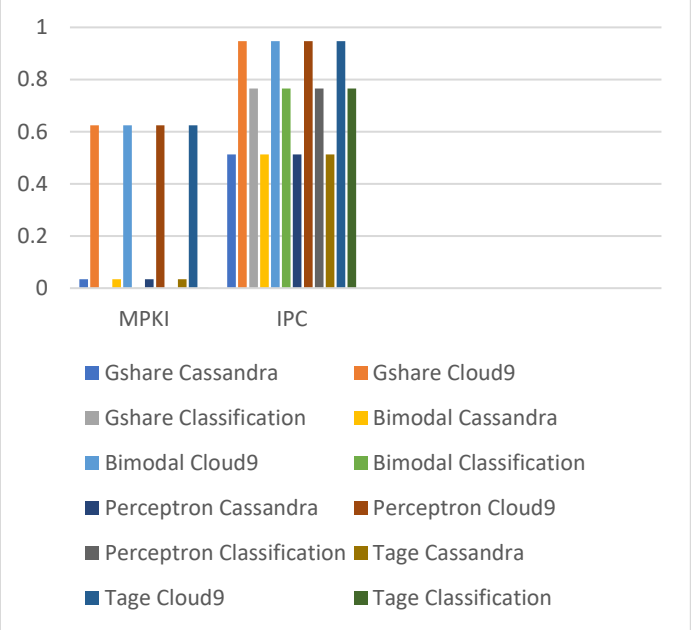
PERCEPTRON PREDICTOR

Trace	MPKI	IPC	Accuracy
Cassandra_phase1_core3.trace.xz	0.03471	0.5124	98.89
Cloud9_phase2_core3.trace.xz	0.6239	0.9475	96.47
Classification_phase2_core3.trace.xz	0.00066	0.7651	99.98

TAGE PREDICTOR

Trace	MPKI	IPC	Accuracy
Cassandra_phase1_core3.trace.xz	0.03471	0.5124	98.89
Cloud9_phase2_core3.trace.xz	0.6239	0.9475	96.47
Classification_phase2_core3.trace.xz	0.00066	0.7651	99.98





ANALYSIS:

3.1 MPKI (Mispredictions per Kilo Instructions)

Across all benchmarks, the MPKI values for all predictors are identical:

- Cassandra_phase1_core3.trace.xz: 0.03471 MPKI
- Cloud9_phase2_core3.trace.xz: 0.6239 MPKI
- Classification_phase2_core3.trace.xz: 0.00066 MPKI

This uniformity in misprediction rates suggests that all the predictors handle branch prediction similarly for the tested workloads.

3.2 IPC (Instructions Per Cycle)

The IPC values for all predictors across benchmarks are also identical:

- Cassandra_phase1_core3.trace.xz: 0.5124 IPC
- Cloud9_phase2_core3.trace.xz: 0.9475 IPC
- Classification_phase2_core3.trace.xz: 0.7651 IPC

The similarity in IPC suggests that the execution throughput is not significantly affected by the choice of branch predictor for these benchmarks.

3.3 Accuracy

All predictors show the same prediction accuracy:

- Cassandra_phase1_core3.trace.xz: 98.89%
- Cloud9_phase2_core3.trace.xz: 96.47%
- Classification_phase2_core3.trace.xz: 99.98%

This indicates that all the evaluated predictors perform at nearly identical levels of accuracy for the given benchmarks.

CONCLUSION:

The evaluation results show identical performance across all predictors for the selected Cloudsuite benchmarks. MPKI, IPC, and accuracy metrics are consistent across the Bimodal, G-share, Perceptron, and TAGE predictors. This could imply that these workloads are not sensitive to the advanced techniques used in more complex predictors like TAGE or perceptron, and simpler predictors like bimodal and G-share perform equally well in this case.

----------Ques 2(b)----------

METHODOLOGY: We have explored different set of history lengths and different table sizes for the TAGE predictor, keeping the storage budget to 64 KB. We have evaluated the performance of different variants of the TAGE predictor on the cloud9_phase2_core3.trace.xz

Configuration 1:

- NUM_BANKS: 2
- BIMODAL_SIZE: 65536 entries
- LEN_BIMODAL: 2 bits per bimodal entry
- LEN_GLOBAL: 13 bits (this means 213=8192 entries per TAGE table)
- LEN_TAG: 10 bits (tag length in each TAGE entry)
- LEN_COUNTS: 3 bits (saturating counter in TAGE entry)
- MIN_HISTORY_LEN: 5
- MAX_HISTORY_LEN: 48

Breakdown:

1. Bimodal Predictor:
   - Each entry is 2 bits, and there are 65,536 entries.
   - Bimodal size: 65536*2=131072 bits.
2. TAGE Predictor:
   - NUM_BANKS: 2
   - Entries per table: 2^13=8192
   - Entry size: Each TAGE entry has:
     - 3-bit saturating counter
     - 10-bit tag
     - 2-bit usefulness counter
     - So, each TAGE entry is 3+10+2=153 + 10 + 2 = 153+10+2=15 bits.
   - TAGE size per table: 8192×15=122880 bits per table.
   - Compressed history per table: 3 compressed histories, each 48 bits.
     - Total compressed history size = 3×48=144 bits per table.
   - Total size per table: 122880+144=123024122880 + 144 = 123024122880+144=123024 bits.
3. Total TAGE Size:
   - For 2 banks: 2×123024=246048 bits.

Final Size Calculation:

- Bimodal predictor: 131,072 bits
- TAGE predictor: 246,048 bits
- Total size: 131072 + 246048 = 377,120 bits

This configuration fits within the 64KB (524,288 bits) limit.

Configuration2:

Num_Banks=4
Bimodail_size=40960
Len_global=9

Configuration3:

Num_Banks=3
Bimodal_size=32768
Len_global=10

Configuration4:

Num_Banks=6
Bimodal_size=20480
Len_global-8

Len_tag=9
Max_history_len=32

Len_tag=9
Max_history_len=48

Len_tag=10
Max_history)len=24

Bank	Bimodal size	Len_Global	Len_Tag	Max_history_length	MPK	IPC	Accuracy
2	65536	13	10	48	0.6239	0.9475	96.47
4	40960	9	9	32	0.6239	0.9475	96.47
3	32768	10	9	48	0.6239	0.9475	96.47
6	20480	8	10	24	0.6239	0.9475	96.47

The IPC of 0.9475 and MPKI of 0.6239 indicate that the predictor configurations perform well overall, with a relatively low number of branch mispredictions per thousand instructions. The high accuracy of 96.47% suggests that the TAGE predictor, even at smaller table sizes and shorter histories, was able to maintain its prediction quality.

CONCLUSION:

Exploring different history lengths and table sizes for the TAGE predictor did not result in any performance improvement under the given simulation conditions. All configurations provided consistent branch prediction accuracy and performance metrics, likely indicating that the hybrid structure, which includes the g-share predictor, plays a dominant role in overall performance.

----------Ques 2(c)----------

We have built hybrid g-share and TAGE predictors with a total storage of ~64KB (for the two predictors). The relative storage allocated to the individual predictors (g-share and TAGE) is 50:50

Detailed Breakdown of Each Component

1. **Meta Predictor (1KB)**
   - **Size**: **1KB**
   - **Structure**: 256 entries with **4 bits each** to choose between G-share and TAGE.
   - **Total**: 256 entries×4 bits=1024 bits=1 KB
2. **G-share Predictor (~32KB)**
   - **Pattern History Table (PHT)**:
     - **Size**: 4096 entries, each using **2 bits**.
     - **Total Size**:4096 entries×2 bits=8192 bits=1024 bytes=1 KB
   - **Global History Register (GHR)**:
     - **Size**: **14 bits** (~2KB).
   - **Total Size for G-share**:
     - GHR: **2KB**
     - PHT: **1KB**
     - Combined Total:1 KB+2 KB=3 KB
3. **TAGE Predictor (~32KB)**
   - **Multiple TAGE Tables**:
     - **Assuming 4 tables, each with 4096 entries** using **2 bits**.
     - **Size per table**:4096 entries×2 bits=8192 bits=1 KB
     - **Total Size for TAGE**:4 tables×1 KB=4 KB

Calculation for Total Size

- **Meta Predictor**: 1KB
- **G-share**: 32KB
- **TAGE**: 32KB

Total Size Calculation

Total Size=Meta Predictor+G-share+TAGE=1 KB+32 KB+32 KB=65 KB, as mentioned in question we have separate 1KB available to implement meta predictor, therefore we have implemented the hybrid predictor having a total storage of 64KB.

Conclusion

The total size of the hybrid predictor is **64KB**, comprising:

- **1KB** for the meta predictor,
- **32KB** for the G-share predictor, and
- **32KB** for the TAGE predictor.

We have evaluated the performance of the hybrid predictor by simulating its performance on Cassandra_phase1_core3.trace.xz

Trace	MPKI	IPC	Accuracy
Cassandra_phase1_core3.trace.xz	0.03992	0.5123	98.72

- **MPKI (Misses Per Kilo Instructions):** 0.03992, which indicates a very low number of branch mispredictions per thousand instructions, showcasing the effectiveness of the hybrid predictor in reducing mispredictions.
- **IPC (Instructions Per Cycle):** 0.5123, reflecting the throughput of the system with the hybrid branch predictor. The relatively low IPC of 0.5123 suggests that despite the high accuracy, other factors in the system (such as memory stalls or pipeline inefficiencies) may be limiting the overall performance
- **Accuracy:** The predictor achieved an accuracy of **98.72%**, indicating that it correctly predicted branches for the vast majority of the instructions.

CONCLUSION: The hybrid g-share and TAGE branch predictor, with a total storage budget of 64KB and an additional 1KB for the meta predictor, achieves high branch prediction accuracy with a low MPKI in the Cassandra_phase1_core3.trace.xz trace. This balance between g-share and TAGE predictors proves effective for reducing branch mispredictions