E0243: High Performance Computer Architecture

Assignment 1

Group ID - G20

Akash Maji SR - 24212 akashmaji@iisc.ac.in

Nehal Shah SR - 24623 nehalshah@iisc.ac.in

September 30, 2024

Question 1: Comparison of Performance in Matrix multiplication program

In this experiment, we utilize **perf**, a widely-used Linux performance measurement tool, to monitor various Performance Monitoring Unit (PMU) events. These events include both hardware-based metrics (such as cache misses at different levels: L1, L2, L3) and software-based metrics (such as page faults and TLB misses). perf allows us to gain insights into system behavior while executing computationally intensive operations.We analyze the performance of a standard $O(n^3)$ matrix multiplication algorithm, implemented in C/C++. Matrix multiplication is inherently compute-bound, making it an ideal candidate to evaluate system-level performance events like cache utilization and memory access patterns.
The matrix multiplication is executed in three different loop orders:

- **(i, j, k)**: iterate first over *i*, then *j*, and finally *k*.
- **(j, i, k)**: iterate first over *j*, then *i*, and finally *k*.
- **(k, i, j)**: iterate first over *k*, then *i*, and finally *j*.

Part (a): Basic Versions with 4KB Pages

In this part, we measure the performance of the matrix multiplication program for the three loop orders using standard 4KB memory pages. This is the default memory page size on most systems, and we record events such as cache misses and page faults under these conditions.

Metric	**(i,j,k)**		**(j,i,k)**		**(k,i,j)**	
	2048	**8192**	**2048**	**8192**	**2048**	**8192**
Duration time (ns)	**30.839,546,165**	**3,657.000,821,699**	**28.747,145,839**	**4,426.857,946,943**	**24.600,546,672**	**1,578.107,532,743**
User time (ns)	**30.810,343,000**	**3,656.602,950,000**	**28.713,868,000**	**4,425.380,384,000**	**24.579,683,000**	**1,577.534,617,000**
Page-faults	**12,376**	**196,804**	**12,377**	**197,230**	**12,377**	**196,808**
L1-dcache-loads	**180,716,201,859**	**11,548,025,757,637**	**180,799,591,141**	**11,546,530,116,698**	**180,493,924,548**	**1,548,034,424,737**
L1-dcache-load-misses	**13,051,193,798**	**1,200,446,330,014**	**12,931,597,982**	**1,175,863,490,126**	**566,698,299**	**68,981,574,642**
L1-dcache-stores	**17,324,396,716**	**1,102,180,578,461**	**17,323,845,706**	**1,102,123,802,619**	**17,280,379,677**	**1,101,538,256,169**
L1-icache-load-misses	**4,028,646**	**367,988,829**	**4,192,323**	**523,788,201**	**4,980,158**	**193,654,440**
LLC-loads	**717,213,971**	**963,041,823,085**	**97,421,469**	**957,922,755,311**	**10,044,197**	**294,832,978**
LLC-load-misses	**20,139,255**	**1,305,607,741**	**10,706,600**	**2,861,865,612**	**6,827,789**	**205,877,525**
LLC-stores	**681,560**	**51,369,940**	**663,832**	**44,348,743**	**579,777**	**24,475,056**
LLC-store-misses	**285,538**	**15,393,320**	**286,705**	**18,226,105**	**321,250**	**13,983,975**
dTLB-loads	**180,522,694,359**	**11,550,454,804,049**	**180,650,953,750**	**11,552,726,081,292**	**180,800,981,910**	**1,548,569,857,358**
dTLB-load-misses	**3,124,469,564**	**543,022,254,070**	**2,985,825,485**	**518,437,302,979**	**6,763,029**	**129,763,674**
dTLB-stores	**17,290,052,333**	**1,102,257,021,574**	**17,305,298,422**	**1,102,405,097,230**	**17,344,176,075**	**1,101,875,244,138**

dTLB-store-misses	**111,706**	**9,517,610**	**625,479**	**20,618,786**	**296,218**	**9,327,580**
iTLB-load-misses	**81,775**	**10,251,533**	**63,775**	**13,713,203**	**45,773**	**1,392,163**
Branch-instructions	**8,782,139,839**	**76,749,309**	**4,557,846**	**85,614,929**	**4,621,447**	**75,776,978**
Branch-misses	**4,576,725**	**553,185,083,901**	**8,775,284,817**	**553,363,279,103**	**8,774,585,811**	**552,735,244,961**
Bus-cycles	**730,209,252**	**86,486,454,981**	**681,141,936**	**104,104,517,082**	**575,552,191**	**37,048,955,948**

Table 1:Performance measure of program for standard pages.

When analyzing the efficiency of iteration orders with normal 4KB pages, the order of efficiency in terms of time consumption is as follows: **[I, J, K] >[J, I, K] > [K, I, J]**. This means that the iteration order is the [K, I, J] least time-consuming, followed by [J, I, K], while [I, J, K] takes the most time.

For the **2048 variants**, the **(k,i,j)** iteration order proves to be the fastest, largely due to the fact that it incurs the fewest LLC (Last Level Cache) stores and loads, which minimizes data retrieval time from memory. On the other hand, the **(i,j,k)** order is the slowest as it incurs the most LLC stores and loads, leading to higher memory access latency.

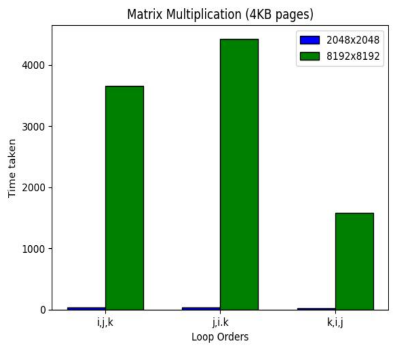


Figure 1:comparison on Time elapsed for standard pages.

For the **8192 variants**, the **(k,i,j)** iteration pattern remains the most efficient, primarily because it experiences the fewest LLC store-misses and load-misses. Store-misses and load-misses occur when the required data is not present in the cache, which results in additional memory access time. In contrast, the **(j,i,k)** order experiences the highest number of store-misses and load-misses, making it the slowest variant for 8192 iterations.

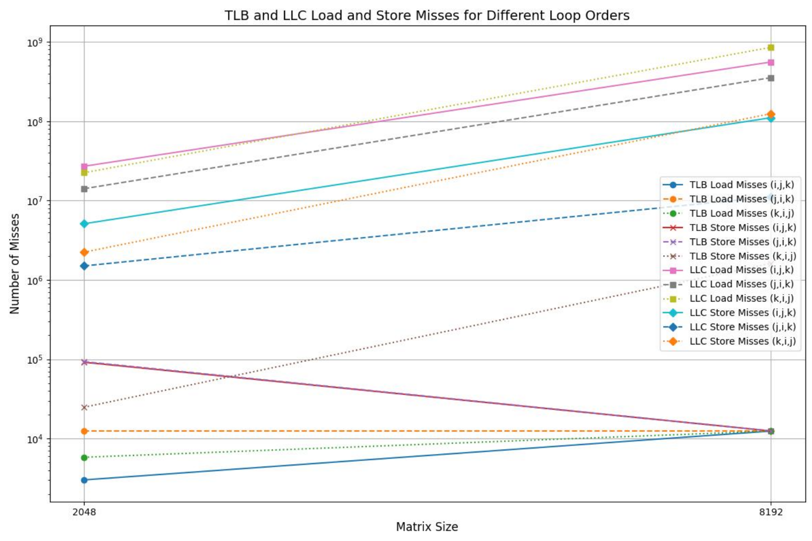


Figure 2:comparison of TLB and LLC misses for standard pages.

Across both the **2048** and **8192** variant groups, all three iteration orders exhibit **almost identical page fault counts**. A page fault occurs when the memory required by the program is not in the physical memory, leading to data being retrieved from disk, which is time-consuming. Despite the differences in cache behavior and time consumption, page faults are consistent across these iteration orders for both 2048 and 8192 groups.

Part (b): Basic Versions with 2MB Pages

In this part, we evaluate the performance of matrix multiplication using 2MB huge pages for two matrix sizes: 2048×2048 and 8192×8192. The 2048×2048 matrix comprises 2^{22} integers, requiring 16MB and necessitating 8 huge pages per matrix, totaling 24 huge pages for three matrices(A, B, and C). Conversely, the 8192×8192 matrix contains 2^{26} integers, which demands 56MB and requires 128 huge pages per matrix, resulting in a total of 384 huge pages for three matrices. To manage this memory efficiently, we utilize the **mmap** system call for dynamic allocation, enabling the allocation of memory for matrices with appropriate flags for huge pages.

Metric	**(i,j,k)**		**(j,i,k)**		**(k,i,j)**	
	2048	**8192**	**2048**	**8192**	**2048**	**8192**
Duration time (ns)	**93.711,056,202**	**11,139.637,268,956**	**92.630,806,364**	**11,134.738,841,697.00**	**23.323,075,474**	**1,566.293,575,103**
User time (ns)	**93.704,515,000**	**11,136.436,269,000**	**92.627,513,000**	**11,131.221,227,000**	**23.322,073,000**	**1,565.706,047,000**
Page-faults	**76**	**438**	**76**	**436**	**77**	**437**
L1-dcache-loads	**146,111,309,862**	**9,351,843,396,129**	**146,274,782,105**	**9,351,893,672,538**	**146,346,423,246**	**9,348,517,946,862**
L1-dcache-load-misses	**8,628,169,783**	**554,643,660,292**	**9,166,601,291**	**585,449,138,806**	**552,740,466**	**66,942,935,635**
L1-dcache-stores	**17,277,682,693**	**1,103,796,102,446**	**17,308,685,726**	**1,103,650,129,000**	**17,315,062,334**	**1,101,331,384,271**
L1-icache-load-misses	**8,293,170**	**1,254,965,004**	**9,859,822**	**1,277,390,048**	**3,207,756**	**217,260,196**
LLC-loads	**2,701,304,319**	**550,138,866,918**	**2,913,869,006**	**550,133,261,235**	**5,327,420**	**213,379,416**
LLC-load-misses	**1,139,672,283**	**548,435,729,154**	**954,685,548**	**548,151,801,003**	**4,289,713**	**166,808,666**
LLC-stores	**3,910,346**	**582,761,944**	**6,287,668**	**211,207,525**	**1,078,413**	**24,755,121**
LLC-store-misses	**2,203,642**	**522,126,714**	**2,707,380**	**101,606,528**	**766,603**	**10,942,586**
dTLB-loads	**146,428,902,934**	**9,353,519,539,316**	**146,317,418,345**	**9,353,470,761,736**	**146,124,811,633**	**9,346,937,597,114**
dTLB-load-misses	**3,012**	**13,429,387**	**12,430**	**18,071,320**	**5,810**	**829,690**
dTLB-stores	**17,325,900,270**	**1,103,765,705,725**	**17,301,784,980**	**1,103,699,301,055**	**17,284,801,579**	**1,101,350,107,039**
dTLB-store-misses	**91,607**	**11,835,387**	**93,163**	**12,470,670**	**24,827**	**1,618,156**
iTLB-load-misses	**7,877**	**1,740,466**	**15,404**	**2,558,554**	**6,010**	**367,316**
Branch-instructions	**4,613,776**	**99,854,918**	**4,683,516**	**102,508,848**	**4,558,781**	**76,515,626**
Branch-misses	**8,760,926,085**	**554,780,482,096**	**8,767,129,620**	**554,820,377,522**	**8,759,089,629**	**552,441,645,258**
Bus-cycles	**2,234,646,940**	**265,443,790,303**	**2,211,446,148**	**265,344,609,240**	**545,669,830**	**36,795,383,547**

Table 2:Performance measures of program for Huge pages.

When using huge 2MB pages, the order of efficiency in terms of time consumption is: **[I, J, K] > [J, I, K] > [K, I, J]**, with **[K, I, J]** being the most efficient and**[I, J, K]** the least.

For the **2048 variants**, the **(k,i,j)** iteration order is the fastest, primarily because it experiences the fewest LLC (Last Level Cache) stores and loads. On the other hand, **(i,j,k)** takes the most time, as it incurs the highest number of LLC load-misses, leading to more memory retrievals from outside the cache, which slows down performance.

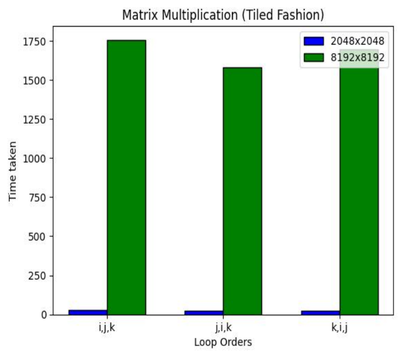


Figure 3:comparison on Time elapsed for Huge pages.

Similarly, for the **8192 variants**, **(k,i,j)** is again the fastest, thanks to the minimal LLC store-misses and load-misses. In contrast, the **(j,i,k)** order is the slowest in this case because it incurs the most LLC stores and store-misses, resulting in frequent memory access delays.

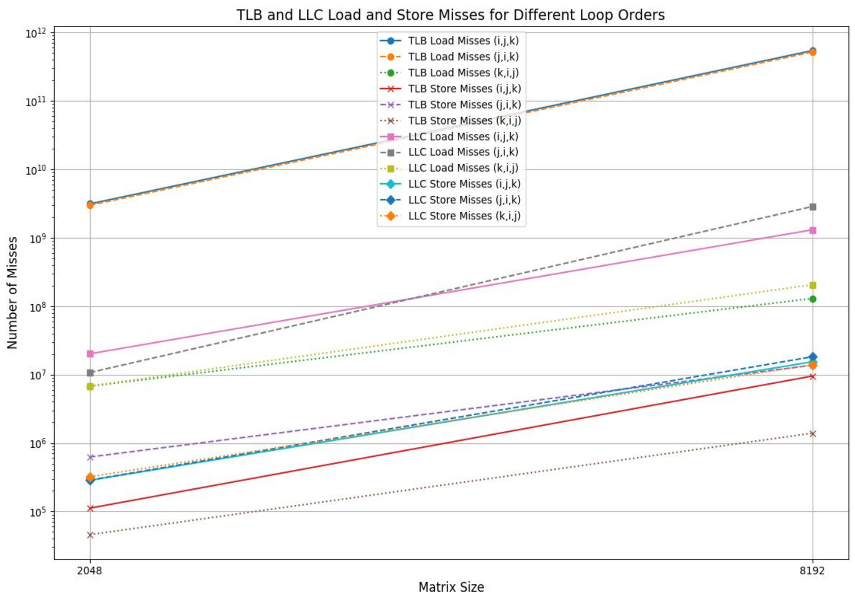


Figure 4:comparison of TLB and LLC misses for Huge pages.

Across both the 2048 and 8192 variant groups, all three iteration orders exhibit **almost identical page fault counts**, indicating consistency in memory management behavior across the different iteration patterns, despite varying cache efficiency and time consumption.

Part (c): Tiled Versions with 4KB Pages

In this part, we introduce tiling (blocking) to optimize matrix multiplication for better cache locality.Tiling reduces cache misses by dividing the matrix into smaller blocks, allowing more of the matrix to fit into faster cache memory during operations. We measure PMU events again using 4KB pages for the tiled versions, comparing the impact of tiling against the basic implementations.

Metric	**(i,j,k)**		**(j,i,k)**		**(k,i,j)**	
	2048	**8192**	**2048**	**8192**	**2048**	**8192**
Duration time (ns)	**27.941,030,455**	**1,756.026,040,607**	**25.647,599,198**	**1,692.886,438,544**	**24.635,800,144**	**1,579.586,545,771**
User time (ns)	**27.920,088,000**	**1,755.841,046,000**	**25.627,140,000**	**1,692.689,712,000**	**24,623,036,000**	**1,579,331,811,000**

Page-faults	12,377	196,805	12,377	196,827	12,376	196,808
L1-dcache-loads	190,194,232,622	12,150,517,050,774	190,010,392,317	12,149,808,482,668	190,260,678,597	12,149,874,528,789
L1-dcache-load-misses	61,483,215	3,446,557,830	318,199,290	19,942,640,700	65,210,685	3,672,203,829
L1-dcache-stores	17,576,630	1,118,981,561,228	17,556,658,794	1,118,977,407,534	17,594,147,665	1,118,889,954,245
L1-icache-load-misses	4,857,490	149,885,643	3,429,295	151,370,928	3,586,939	144,493,229
LLC-loads	4,731,392	371,554,980	6,612,223	517,173,439	1,576,002	83,369,550
LLC-load-misses	2,325,423	283,490,200	553,649	404,442,395	525,688	29,758,027
LLC-stores	666,641	24,699,157	603,218	23,768,203	597,245	21,997,877
LLC-store-misses	417,711	11,171,659	398,225	11,933,107	408,272	11,181,211
dTLB-loads	189,938,841,938	12,151,094,554,189	189,978,432,782	12,151,335,842,639	189,246,743,067	12,153,672,025,760
dTLB-load-misses	2,018,513	143,697,521	1,980,500	143,434,123	187,679	3,635,874
dTLB-stores	17,581,663,757	1,118,892,390,497	17,565,664,488	1,118,934,040,459	17,525,520,826	1,118,969,231,880
dTLB-store-misses	95,280	3,263,733	90,075	3,251,544	91,484	3,224,840
iTLB-load-misses	17,707	185,199	5,703	206,662	3,867	116,615
Branch-instructions	136,791,425	8,732,335,847	136,695,025	8,730,766,311	136,601,847	8,717,341,208
Branch-misses	9,175,329,258	578,749,548,881	9,168,421,057	578,636,484,224	9,169,744,340	578,559,572,924
Bus-cycles	660,603,499	41,737,917,678	610,683,266	39,853,243,827	583,186,121	37,105,852,844

Table 3:Performance measures of program for Tiled version of standard pages.

In this scenario, the time savings across the three loop orders are minimal, with only slight differences in the time taken for each iteration pattern.

For the **2048 variant**, the **(i,j,k)** loop order incurs the highest number of LLC load-misses and stores, making it the least efficient. In contrast, **(k,i,j)** performs the best, with the fewest LLC load-misses and stores.

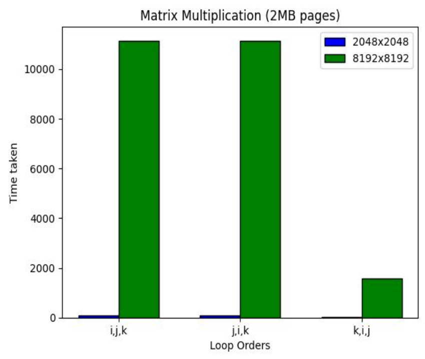


Figure 5:comparison on Time elapsed for Huge pages.

Similarly, for the **8192 variant**, **(i,j,k)** again shows the worst performance, incurring the most LLC stores, while **(k,i,j)** is the most efficient, with the least LLC stores.

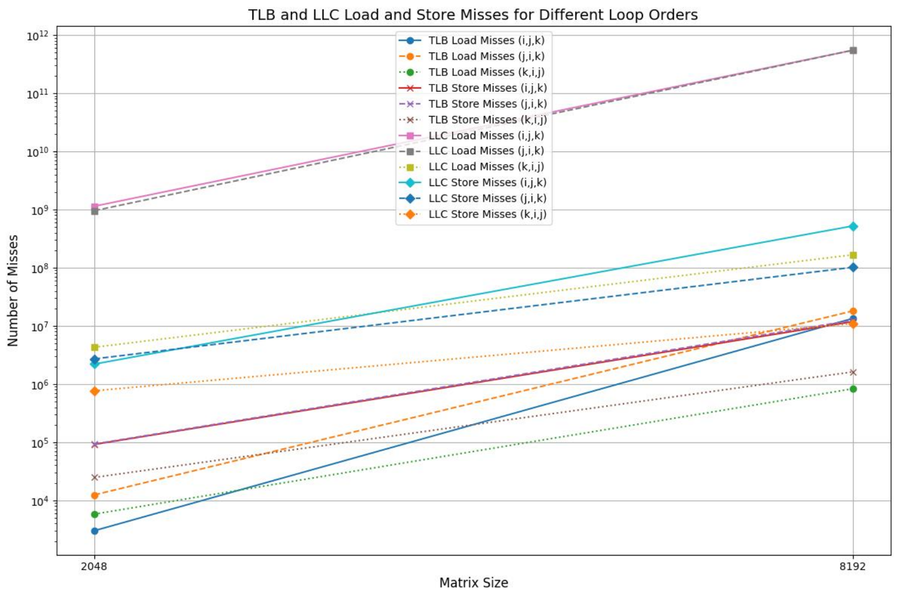


Figure 6:comparison of TLB and LLC misses for Tiled version of standard pages.

Overall, **(i,j,k)** emerges as the least efficient order, performing poorly in both the 2048 and 8192 cases, similar to its performance in non-tiled versions. On the other hand, **(k,i,j)** continues to take the least amount of time, proving to be the fastest iteration order.

Question 2: Cost-effective Branch Predictors

We are conducting simulations using ChampSim with various branch predictor variants , utilizing traces from the CloudSuite and SPEC benchmark. The three specific **Cloudsuite** traces and two **SPEC** traces being used for the simulations are referred to as **Trace 1** ("nutch_phase2_core1.trace.xz"), **Trace 2** ("cloud9_phase1_core1.trace.xz"), **Trace 3** ("classification_phase0_core3.trace.xz"), **Trace 4** ("600.perlbench_s-210B.champsimtrace.xz"), **Trace 5** ("602.gcc_s-734B.champsimtrace.xz"). ChampSim is an open-source trace-based simulator maintained at Texas AM University, which allows users to integrate and simulate their own branch predictors, prefetchers, and cache replacement policies. **For each simulation, we employ a 10-million instruction warm-up phase followed by 500 million instructions for the actual simulation**. The goal of these simulations is to observe and analyze key performance parameters influenced by different branch predictor designs. By experimenting with these predictor variants on the specified CloudSuite traces, we aim to gain insights into their performance characteristics under diverse workloads.

Part (a): Branch Predictor Performance Comparison

We run the benchmarks on the four aforementioned branch predictions in Champsim with the hardware restriction of 64KB. We use the already available implementations of *GShare* and *Perceptron* predictors in ChampSim. The TAGE predictor is implemented using the open source implementation.

Storage Justification (64 KB)

Bimodal Predictor Storage Cost Estimation: For the bimodal predictor, we aim to use at most 64KB per CPU. Utilizing a 2-bit saturating counter, each entry size is 2 bits. The number of entries is calculated as 64KB / 2 bits, which equals 64 * 8 Kbits / 2 bits = 256 Kbits = 262144 entries. The largest prime number less than 262144 is 262139. Therefore, the bimodal table size is defined as 262144, with a prime number of 262139 and a maximum counter value of 3.

GShare Predictor Storage Cost Estimation: For the GShare predictor, we also aim to use at most 64KB per CPU. Using a 2-bit saturating counter, each entry size is 2 bits. The number of entries is calculated as 64 * 8 Kbits / 2 bits = 256 Kbits = 262144 entries. The global history length is defined as 16, with a corresponding mask.

Perceptron Predictor Storage Cost Estimation: Each perceptron consists of 32 weights, with each weight occupying 8 bits, resulting in each perceptron being 32 bytes. The table contains 2048 perceptron, making the Perceptron Table Size 2048 * 32 bytes, which equals 64 KB. The Update Table Entry Size is 8 bytes at maximum, calculated as 32 + 1 + 1 + 11 = 6 bytes. With 256 entries, the Update Table Size is 256 * 6 bytes, which is less than 2 KB. Therefore, the total size taken is within 64 + 2 KB.

TAGE Predictor Storage Budget Justification: The TAGE predictor uses 12 tables with entry sizes of 24, 22, 20, 20, 18, 18, 16, 16, 14, 14, 12, and 12 bits. The tag widths are 19, 17, 15, 15, 13, 13, 11, 11, 9, 9, 7, and 7 bits, with 5 bits allocated for the 'ctr' and 'u' fields. Each TAGE table has 2^{11} entries, making the total TAGE Tables Size (sum of entry sizes * 2^{11}) equal to 206 bits * 2048, which is 52 KB. The Bimodal Table Size, with 2^{13} entries each of 2 bits, is 2^{14} bits or 2^{11} bytes, equating to 4 KB. Thus, the total size is 52 KB + 4 KB, which fits within the 64 KB budget.

Trace	MPKI	IPC	Prediction Accuracy(%)	Predictor Used
Trace2	4.374	0.393449	97.99	Bimodal
	0.73988	0.409175	99.66	GShare
	0.416594	0.410222	99.81	Perceptron
	0.117964	0.409526	99.95	TAGE

(a) Benchmark metrics for Trace2

Trace	MPKI	IPC	Prediction Accuracy(%)	Predictor Used
Trace1	16.5163	1.36976	93.50	Bimodal
	0.005922	2.05911	100.00	GShare
	0.00844	2.059	100.00	Perceptron
	0.006368	2.05922	100.00	TAGE

(b) Benchmark metrics for Trace1

Trace	MPKI	IPC	Prediction Accuracy(%)	Predictor Used
Trace3	4.42255	0.361737	96.69	Bimodal
	3.08063	0.364767	97.69	GShare
	2.48727	0.366098	98.14	Perceptron
	1.6329	0.370232	98.78	TAGE

(c)Benchmark metrics for Trace 3

Table 4:Branch predictor results for all three Cloudsuite Benchmarks.

- **Trace 1 (nutch_phase2_core1.trace.xz):**

The TAGE predictor delivers the highest Instructions Per Cycle (IPC), indicating that it is the most efficient in predicting branches correctly.On the other hand, the BIMODAL predictor exhibits the lowest IPC, demonstrating that it struggles with branch prediction accuracy for this trace.

The TAGE predictor also achieves the lowest Misses Per Thousand Instructions (MPKI), which shows that it generates fewer mispredictions per thousand instructions.In contrast, the BIMODAL predictor has the highest MPKI, signifying more frequent mispredictions, leading to a negative impact on overall performance.

- **Trace 2 (cloud9_phase1_core1.trace.xz):**

In this trace, the Perceptron predictor surpasses other predictors by yielding the highest IPC, demonstrating its strong branch prediction capabilities for this specific workload.Once again, BIMODAL presents the lowest IPC, indicating that it performs poorly in terms of prediction accuracy.

Similar to Trace 1, the TAGE predictor achieves the lowest MPKI, indicating the fewest mispredictions and overall better performance in terms of accuracy.Conversely, the BIMODAL predictor produces the highest MPKI, demonstrating higher misprediction rates.

- **Trace 3 (classification_phase0_core3.trace.xz):**

The TAGE predictor continues its strong performance with the highest IPC, showcasing its effectiveness in branch prediction for this trace.The BIMODAL predictor again demonstrates the lowest IPC, reinforcing its limited effectiveness in accurately predicting branches for this workload.

The TAGE predictor secures the lowest MPKI, indicating its superior accuracy in branch prediction with fewer misdirection.In contrast, the BIMODAL predictor records the highest MPKI, further confirming its tendency to make more mispredictions.

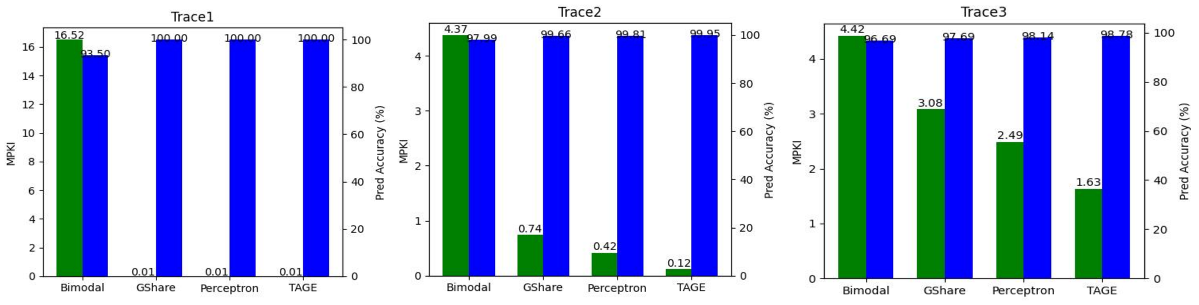


Figure 7:Graph on MPKI and Accuracy vs branch predictor for all Cloudsuite Benchmark.

- **Trace 4 (600.perlbench_s-210B.champsimtrace.xz):**

The **TAGE** predictor yields the highest Instruction Per Cycle (IPC), indicating superior efficiency compared to the other predictors. On the contrary, the **Bimodal** predictor records the lowest IPC, reflecting a less efficient performance for this trace. When considering Misses Per Kilo Instructions (MPKI), **TAGE** achieves the lowest value, which demonstrates its ability to minimize prediction errors effectively, while the **Bimodal** predictor produces the highest MPKI, suggesting a higher rate of mispredictions.

Trace	MPKI	IPC	Prediction Accuracy(%)	Predictor Used
Trace 4	3.56801	1.3304	97.505	Bimodal
	2.42522	1.3883	98.3041	GShare
	1.77504	1.42387	98.7588	Perceptron
	1.06583	1.47161	99.2547	TAGE

(d)Benchmark metrics for Trace 4

Trace	MPKI	IPC	Prediction Accuracy(%)	Predictor Used
Trace 5	12.6206	0.64183	94.707	Bimodal
	0.496634	0.770182	99.7917	GShare
	0.473712	0.770465	99.8013	Perceptron
	0.190268	0.774499	99.9202	TAGE

(e) Benchmark metrics for Trace 5

Table 5:Branch predictor results for all two SPEC Benchmarks.

- **Trace 5 (602.gcc_s-734B.champsimtrace.xz):**

The **Perceptron** predictor stands out by achieving the highest IPC, indicating it is the most effective at maintaining execution throughput. In contrast, **Bimodal** again results in the lowest IPC, reflecting its relatively poor performance for this trace. In terms of MPKI, **TAGE** leads with the least MPKI, showing it provides the most accurate predictions, while **Bimodal** once more registers the highest MPKI, confirming its weaker performance for trace5.

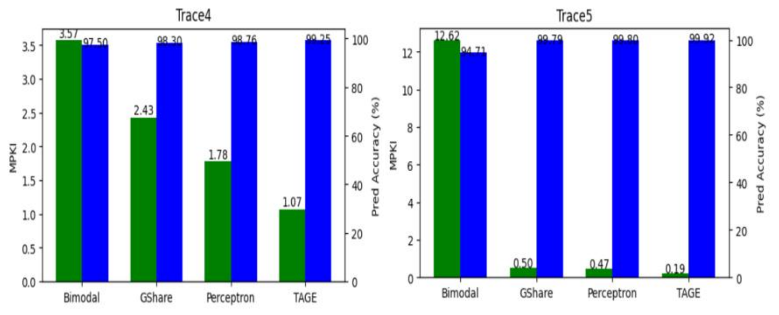


Figure 8:Graph on MPKI and Accuracy vs branch predictor for all SPEC Benchmark.

we have noted that in all 5 traces, as we go from BIMODAL to GSHARE to PERCEPTRON to TAGE, the Prediction Accuracy increases, indicating that mispredictions reduce in that order.

So, **TAGE** predictor has the best outcomes and **BIMODAL** is least desirable.

Part (b): Effect of Varying History lengths and Table sizes

Impact of varying the minimum global History lengths:

we are conducting CloudSuite and SPEC simulations using ChampSim with TAGE predictor variants, focusing on a configuration with 8 TAGE tables. We will first examine the impact of varying the minimum global history lengths across different cases while keeping the maximum table size fixed at 2^{13} entries. we are simulating using 500M instructions.

The global history lengths are adjusted as follows:

- **Min history length 4**: [4, 6, 10, 16, 26, 42, 67, 107]
- **Min history length 8**: [8, 13, 20, 33, 52, 84, 134, 215]
- **Min history length 12**: [12, 19, 31, 49, 79, 126, 201, 322]
- **Min history length 16**: [16, 26, 41, 66, 105, 168, 268, 429]

The number of bits used to index into each of the 8 TAGE tables varies, with table sizes being determined by the index bits: 13, 13, 12, 12, 11, 11, 11, and 11. Similarly, the number of bits for tagging into each table is: 13, 13, 11, 11, 9, 9, 9, and 9. This setup allows us to observe potential improvements in prediction accuracy when varying the history lengths and table sizes for the TAGE predictor.The key observations are listed here -

Trace	4		8		12		16	
	MPKI	IPC	MPKI	IPC	MPKI	IPC	MPKI	IPC
Trace1	0.00743333	2.05839	0.00785	2.05832	0.00795333	2.05831	0.00816667	2.05826
Trace2	0.0963733	0.409516	0.103427	0.409591	0.11343	0.409467	0.114957	0.409534
Trace3	1.92031	0.368561	1.81499	0.369058	1.75186	0.369506	1.69893	0.369608

Table 6: Performance Metrics with varying minimum History Lengths for Cloudsuite Benchmarks.

In the **CloudSuite traces**, the effect of increasing the minimum history length does not follow a consistent pattern. For **trace1** and **trace2**, the **MPKI** increases as the history length grows, indicating a decline in prediction accuracy. However, for **trace3**, the **MPKI** decreases with the increase in history length, showing improved prediction. Additionally, **trace3** shows an increase in **IPC**, highlighting that longer history lengths positively impact both prediction accuracy and throughput for this particular trace, while the other traces behave differently. This variance in behavior suggests that the impact of history length on performance is more workload-dependent in CloudSuite traces.

Trace	4		8		12		16	
	MPKI	IPC	MPKI	IPC	MPKI	IPC	MPKI	IPC
Trace4	1.21837	1.4608	1.06583	1.47161	1.04445	1.47338	1.02881	1.47431
Trace5	0.274326	0.7731	0.190268	0.774499	0.182804	0.774735	0.178196	0.774578

Table 7: Performance Metrics with varying minimum History Lengths for SPEC Benchmarks.

In the **SPEC traces**, it is observed that as the minimum history length increases from 4 to 8, 12, and finally to 16, the **IPC** gradually improves while the **MPKI** decreases. This indicates that increasing the history length positively impacts prediction accuracy, reducing the number of mispredictions and improving the throughput of instructions executed per cycle. The trend suggests that a longer history allows the predictor to make more informed decisions, leading to higher performance in terms of both IPC and MPKI.

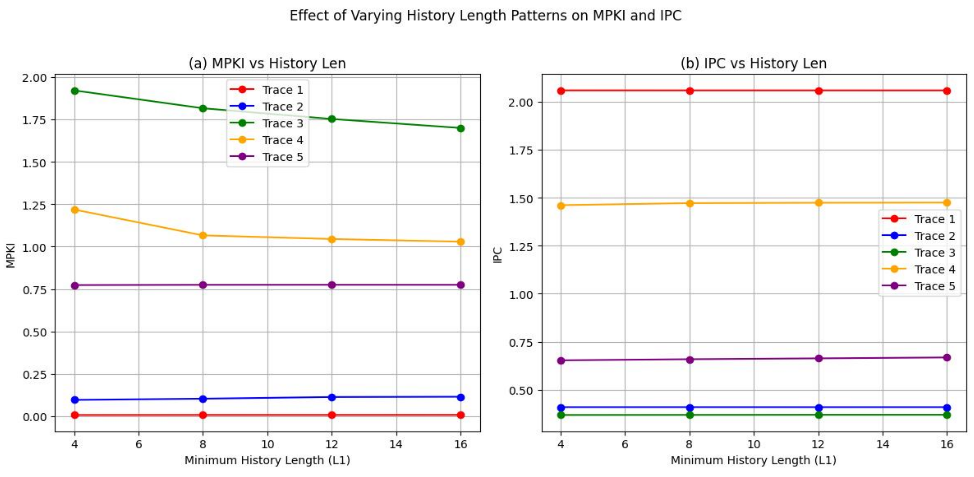


Figure 9:shows the Line Graph for Effect of Varying History Length on TAGE Performance.

Impact of varying the Table sizes:

we explore the impact of using different sizes for TAGE tables while keeping the total storage under 64KB and fixing the history lengths in geometric progression. We evaluate two configurations, **Case A** and **Case B**, with **8 TAGE tables** in each case. The **global history lengths** follow a geometric progression: **[4, 8, 16, 32, 64, 128, 256, 512]**.Comparing these configurations helps determine if adjusting table sizes and tag bits can improve **IPC** and reduce **MPKI**, revealing the trade-offs between entry size and tag precision in a TAGE predictor with fixed history lengths.we simulate with 500M instructions

In **Case A**, each table has **2^12 entries**, with **index bits** set to **12** and **tag bits** decreasing as **{17, 15, 13, 11, 11, 9, 7, 5}**. This configuration balances the number of entries per table with tag precision.

In **Case B**, each table has **2^11 entries**, maintaining **12** index bits but with larger **tag bits**: **{39, 35, 31, 27, 27, 23, 19, 15}**. The larger tags aim for better prediction accuracy through finer distinctions between branches.

Predictor Used	**Trace**	**MPKI**	**IPC**	**Prediction Accuracy**	**MPKI**	**IPC**	**Prediction Accuracy**
TAGE	**Trace1**	**0.004802**	**2.05937**	**99.9981**	**0.006128**	**2.0592**	**99.9976**
	Trace2	**0.207462**	**0.408979**	**99.9047**	**0.113538**	**0.409594**	**99.9478**
	Trace3	**2.50117**	**0.367488**	**98.1278**	**1.88591**	**0.368642**	**98.5884**
	Trace4	**1.74647**	**1.43379**	**98.7787**	**0.988034**	**1.47701**	**99.3091**
	Trace5	**0.259408**	**0.773581**	**99.8912**	**0.181706**	**0.77456**	**99.9238**

Table 8: Performance Metrics with varying Table sizes

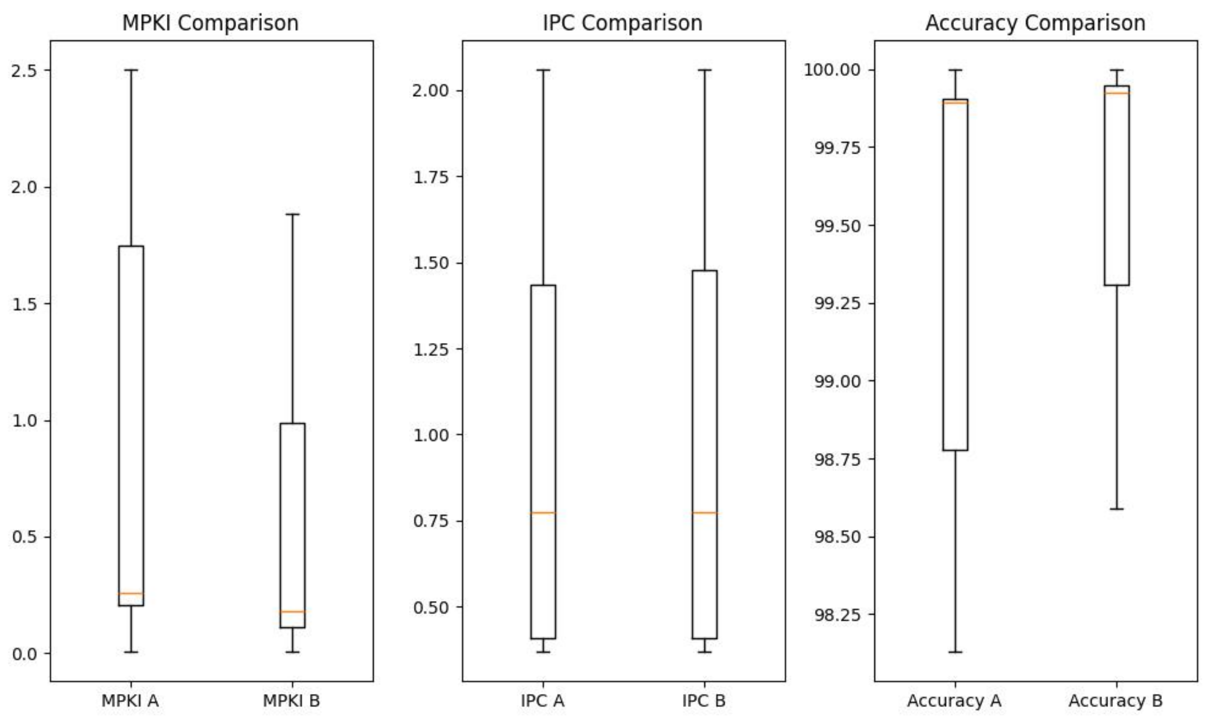


Figure 10:shows the Box Graph for comparison of MPKI, IPC and Accuracy between Case A and Case B.

With **SPEC traces**, we observed that reducing the uniform table size from **2^12** to **2^11 entries** across the **8 TAGE tables** led to a performance improvement. Specifically, this reduction resulted in a higher **IPC** and accuracy, along with a lower **MPKI**. The improvement is attributed to the increase in **TAG sizes** of the tables, allowing for better distinction between branch patterns while maintaining the same **64KB storage budget**.

A similar trend was observed in **trace2** and **trace3** of the **Cloudsuite traces**, where smaller table sizes with longer tags also resulted in better performance. This suggests that, under a fixed storage budget, decreasing table sizes can be beneficial if it allows for longer TAG lengths. It highlights the critical role of TAG lengths in influencing the accuracy and performance of the **TAGE predictor**, showing that sometimes smaller table sizes with more precise tagging can outperform larger tables with shorter tags.

From both configurations of TAGE tables, we see that changing the history lengths results in only **marginal performance differences**. The key observation is the inverse relationship between MPKI and IPC: as MPKI increases, IPC decreases, and vice versa. Despite adjusting history lengths and table sizes, the overall impact on performance remains limited. Significant gains may require more substantial design changes.

Part (c): Hybrid predictor

For our CloudSuite simulations using ChampSim, we are employing a **hybrid predictor** combining **GShare** and **TAGE** while ensuring the total storage remains under **64KB**. The simulation involves varying the storage distribution between the two predictors to observe how different allocations affect performance. Key performance parameters, such as IPC (Instructions Per Cycle), MPKI (Misses Per Kilo Instructions), and predictor accuracy, are analyzed to evaluate the efficiency of this hybrid approach. The goal is to determine the optimal balance between GShare and TAGE within the storage constraints for improved prediction outcomes. We observe the following parameters as stated:

Trace	**Config 25:75**			**Config 50:50**			**Config 75:25**		
	MPKI	**IPC**	**Accuracy**	**MPKI**	**IPC**	**Accuracy**	**MPKI**	**IPC**	**Accuracy**
trace1	**0.0060709 1**	**2.05946**	**100**	**0.00677 091**	**2.05939**	**100**	**0.00769273**	**2.05927**	**100**
trace2	**1.50078**	**0.406576**	**99.31**	**1.51612**	**0.406312**	**99.3**	**1.82682**	**0.404121**	**99.16**

trace3	2.75407	0.362922	97.94	2.85958	0.362615	97.86	3.28599	0.361268	97.54

Table 7:Benchmark results of CloudSuite trace for Hybrid GShare-TAGE Configs

Trace	Config 25:75			Config 50:50			Config 75:25		
	MPKI	IPC	Accuracy	MPKI	IPC	Accuracy	MPKI	IPC	Accuracy
Trace4	2.32977	1.39575	98.3709	2.34259	1.39496	98.3619	2.38446	1.3921	98.3326
Trace5	0.332848	0.772558	99.8604	0.343164	0.772409	99.8561	0.46344	0.770697	99.8056

Table 7:Benchmark results of SPEC Trace for Hybrid GShare-TAGE Configs

- **Config 25:75 (Gshare: 16KB | TAGE: 48KB):**

In this configuration, GShare table has **65,536 entries**, each occupying **2 bits**. For TAGE, we use **8 tables** with varying index lengths to manage different global history lengths, efficiently distributing the storage to capture both short and long histories. This setup maximizes the prediction accuracy within the allocated storage with History lengths {13, 12, 11, 11, 11, 11, 10, 10} and entry sizes for TAGE {9, 10, 11, 12, 12, 13, 14, 15}.

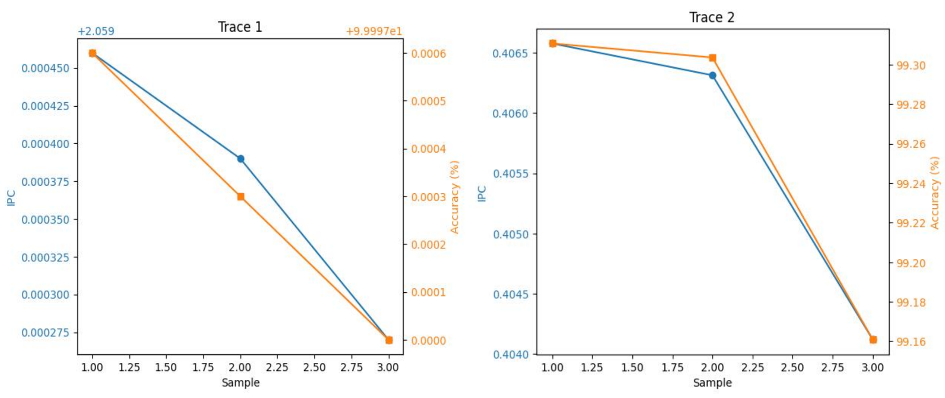
- **Config 50:50 (GShare: 32KB | TAGE: 32KB):**

In this balanced configuration, GShare table consists of **131,072 entries**, each sized at **2 bits**. For TAGE, we utilize **6 tables** with varying index lengths of **12 bits** for the first two tables and **11 bits** for the remaining four. This setup optimizes storage for different global history lengths, enabling precise prediction across a range of scenarios. The History lengths are {12, 12, 11, 11, 11, 10} and entry sizes for TAGE {9, 10, 11, 12, 13, 14}.

- **Config 75:25 (GShare: 48KB | TAGE: 16KB):**

In this configuration, GShare table contains **262,144 entries**, each occupying **2 bits**. For TAGE, we employ **4 tables**, each with index lengths of **11 bits** for the first three tables and **10 bits** for the last one. This design effectively leverages the available storage, allowing for efficient handling of varying workloads while maintaining high prediction accuracy.The History lengths are $\{11, 11, 11, 10\}$ and entry sizes for TAGE $\{11, 12, 13, 14\}$.

Storage justification for 25:75 distribution is given as under. Similarly rest two config can be justified.
Gshare Table Size = 2^16 * 2 bits = 16 KB
Total TAGE Table Size = 2^13 * 14 bits + 2 ^ 12 * 15 bits + … = 44 KB approx
TAGE Bimodal Table Size = 2^14 * 2 bits = 4 KB
Extra 1KB data can be used for a bimodal predictor with 2^12 entries of 2 bits each, as meta predictor



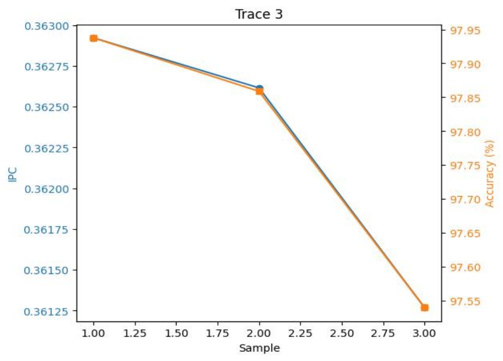


Figure 8:Graphs for Cloudsuite traces showing how IPC and Prediction Accuracy drops as we go from 25:75 to 50:50 to 75:25 configuration.

In the **Cloudsuite** traces (Trace1, Trace2, and Trace3), as we increase the proportion of storage allocated to **GShare** and reduce the storage for **TAGE** across the three hybrid predictor configurations (25:75, 50:50, and 75:25), both **IPC (Instructions Per Cycle)** and **Prediction Accuracy** show a decreasing trend. This means that **TAGE** contributes significantly to the overall performance of the hybrid predictor when given a larger share of the storage budget. As we move from **25:75** to **50:50** and then to **75:25** (where the ratio reflects the storage split between GShare and TAGE), **MPKI (Mispredictions Per Thousand Instructions)** increases. This indicates that **TAGE** is more effective at reducing mispredictions compared to **GShare**. The results suggest that **TAGE** is better suited to handle the dynamic nature of the **Cloudsuite** workloads, and a configuration that favors **TAGE** (like **25:75**) is the best in terms of balancing **IPC** and reducing **MPKI**.

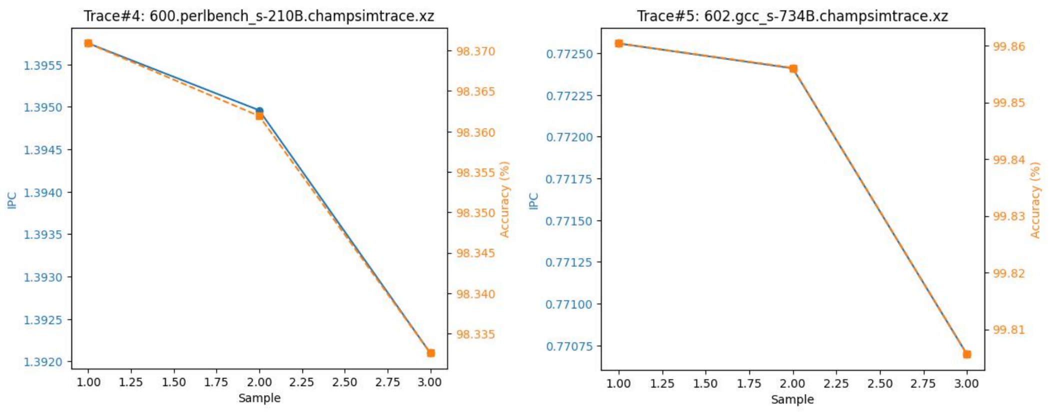


Figure 8:Graphs for SPEC traces showing how IPC and Prediction Accuracy drops as we go from 25:75 to 50:50 to 75:25 configuration.

Similarly, for the **SPEC** traces (Trace4 and Trace5), increasing the fraction of storage to **GShare** and reducing the storage for **TAGE** across the three configurations results in a decrease in both **IPC** and **Prediction Accuracy**. As the storage distribution shifts from **25:75** to **50:50** and further to **75:25**, **MPKI** consistently rises, underscoring the superior predictive capabilities of **TAGE** over **GShare**. The **SPEC** traces, which tend to involve more structured and regular patterns compared to the **Cloudsuite** workloads, still benefit from **TAGE**'s enhanced prediction mechanisms when given a larger share of storage. Thus, a configuration like **25:75** (favoring **TAGE**) is the most optimal, as it allows for better prediction accuracy and **IPC**, while keeping the **MPKI** lower.

CPU Specifications:

Arch	**x86_64**
Model	**Intel(R) core(TM) i5-11500**
CPU Mhz	**4600**
L1d Cache	**288KB**
L1i cache	**192KB**

LLC	**12MB**
Address size	**39 bits physical, 48 bits virtual**

Table 8: CPU specifications

References:

[1] LINUX KERNEL PROFILING WITH PERF. AVAILABLE AT: https://perf.wiki.kernel.org/index.php/Tutorial

[2] LARGE PAGE SUPPORT IN THE LINUX KERNEL: https://linuxgazette.net/155/krishnakumar.html

[3] https://man7.org/linux/man-pages/man2/mmap.2.html

[4] https://man7.org/linux/man-pages/man2/madvise.2.html

[5] CASPERIITB.CHAMPSIM. https://github.com/casperIITB/ChampSim

[6] KANPARD005. RISCY_V_TAGE. https://github.com/KanPard005/RISCY_V_TAGE.

[7] CLOUDSUITE TRACES: https://www.dropbox.com/sh/hh09tt8myuz0jbp/AACAS5zMWHL7coVuS RbpUksa?e=2&from_auth=register&dl=0

[8] SPEC BENCHMARKS: https://dpc3.compas.cs.stonybrook.edu/champsim-traces/speccpu/