CSE240C: LLC Replacement Policy

1st Tanmay Anil Patil
dept. of Electrical and Computer Engineering
University of California, San Diego
San Diego, USA
tapatil@ucsd.edu

***Abstract*—This report evaluates two of the winners (participants) of the Cache Replacement Championship 2 (CRC2) and also performs design space exploration. IPC and MPKI (for LLC) is used for measuring the performance of these policies. 51 traces with the highest MPKI are used for evaluation.**

***Index Terms*—IPC, MPKI, CRC2**

I. Introduction

This report compares the main idea of the top three winners of the Cache Replacement Championship 2 (CRC2), namely, Hawkeye by Akanksha Jain and Calvin Lin, SHiP++ by Vinson Young et al., and ReD by Javier Díaz et al. The top two winners (Hawkeye and SHiP++) are used for design space exploration. The latest version (dated 2/9/2022) of ChampSim is used as the infrastructure with the first configuration (Single core with 2 MB LLC without a prefetcher.) of the CRC2. For every trace the number of warm-up instructions used is 10 million and the statistics is collected on the next 100 million instructions. 51 total traces are selected from the 203 traces available from the CRC2, based on the MPKI (top 3 from each of the 17 categories).
Note: In this report arithmetic mean is used for reporting MPKI and geometric mean is used for reporting IPC.

A. Hawkeye

This paper is a slight modification of "Back to the future: Leveraging Belady's algorithm for improved cache replacement." by A. Jain and C. Lin. The main Idea is to try to mimic Optimal placement/replacement decision. The algorithm has a lower focus on which line to replace, instead it has a very high focus on whether to place the accessed line in cache or not. This decision is taken by OPTgen. It uses a occupancy vector of the cache and priority of the access to decide if it worth placing that line in cache or not. For replacement it uses RRIP values that are initialized when a cache line is placed in the cache. The priority of the cache lines is decided using a predictor that classifies the lines loaded by a given PC as either cachefriendly or cache-averse. Hawkeye improves performance by treating demand accesses and perfetch accesses differently. It tries to avoid cache line that are demand accesses followed by perfetch accesses or that are perfetch accesses followed by perfetch accesses.

B. SHiP++

This paper is a slight modification of the SHiP replacement policy. The main idea of SHiP is to learn the re-reference behaviour of the cache lines and use this knowledge to initialize the RRIP value of the cache line upon placement. SHiP++ improves performance by using various improvements like improved cache insertion, improved SHCT training, writeback-aware RRPV updates, prefetch-aware SHCT training and prefetch-aware RRPV updates.

C. Cache Replacement Policy based on Expected Hit count

Cache hit rate for a block is high is the block has high temporal locality. And so the reverse can also be assume to be true. So the temporal locality for a block will be high is its hit rate is high. This is the main principle used by this policy. It keeps track of the hit rate for every block and at the time of replacement evicts the block with the least hit rate. This data is also used to calculate the expected hit rate on the new accesses and placed in the cache accordingly.

II. Key Design Details

A. Hawkeye

1) Main Data structures: Sampler is used to keep track of various details like if it is a prefetch, the last time it was accessed and liveness intervals and usage interval. It has 2800 entries and needs 4 bytes per entry. The hawkeye predictor has 2k entries with 5-bit counters each. It is instantiated twice, one is used for demand accesses and other is used for prefetch accesses. The occupancy vector (aka OPTgen) has 64 vectors with 128 entries each and each entry is 4 bits. It records the records the occupancy of the cache. The 3-bit RRIP value is used to keep track of the replacement state.

2) Replacement strategy and working: The OPTgen data structure tracks the occupancy of the state. If the occupancy is less than the cache set size then the accessed line is placed in the cache. If it is higher than the cache set size then based on the predictor output (cache-friendly or cache-averse) the cache line is placed in the set. The line to be evicted is decided using RRIP values. A high value mean that this cache line can be replaced and a low value mean it should not be replaced.

B. SHiP++

1) Main Data structures: Signature History Counter Table is used to keep track of the re-reference behaviour of the accesses. It uses signature (generated from PC) to index into it. It has a counter at every entry to keep track of re-references. Similar to Hawkeye, SHiP++ also uses a 2-bit RRIP value to keep track of which cache lines can be evicted. Max value

refers to lowest priority (evict this line first) and zero refers to highest priority (evict this line last).

2) Replacement strategy and working: The SHCT counters keep track of re-references using 3-bit counters at every entry. The signature used to index into this table is last 14 bits of the PC. A high value of the counter indicates that the line is useful and a low value indicated that it is not useful. Useful lines are inserted in the cache with a RRIP value of zero and lines that are not very useful are inserted with a RRIP value of 3. Cache lines are evicted based on the RRIP value. Any cache line with RRIP value of 3 can be evicted. If no cache line in a set is at the max value, the RRIP values for all the cache lines in that set are decremented. And the processes repeats until a cache line is evicted to make space for the new line. The SHCT is only updated on the first hit of a cache line. This avoids cache hit bias and over-training of the SHCT.

III. Reproducing the results

Hawkeye and SHiP++ LLC replacement policy were tested on top 3 traces of each trace category. The top 3 traces were decided on the basis of the MPKI values of the traces.

	MPKI	IPC
LRU	17.93618588	0.61057015
Hawkeye	18.89216725	0.66416129
SHiP++	18.88401843	0.66898342

Improvement	Reproduced	Actual	Error
Hawkeye	8.77%	4.5%	4.27%
SHiP++	9.56%	6.2%	3.36%

We can see from the above tables that the error in the improvement values is high which can be attributed to the fact that we are using only a small subset of the traces that were actually used.

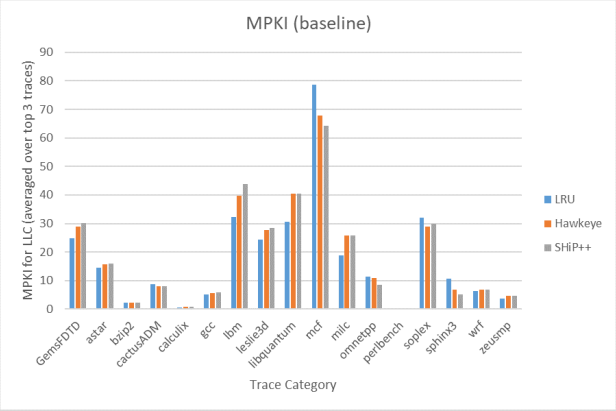

Fig. 1. MPKI for different trace Categories

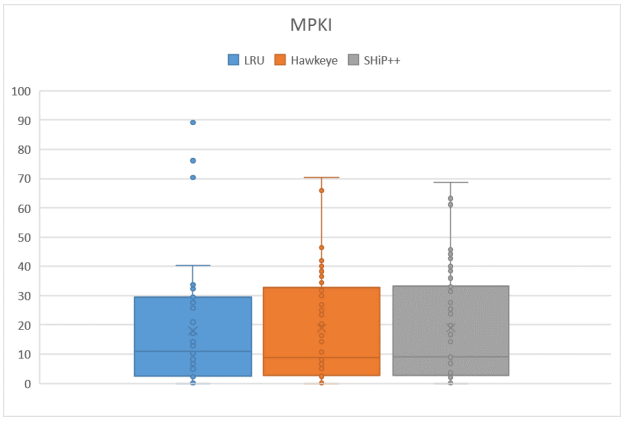

Fig. 2. IPC for different trace Categories

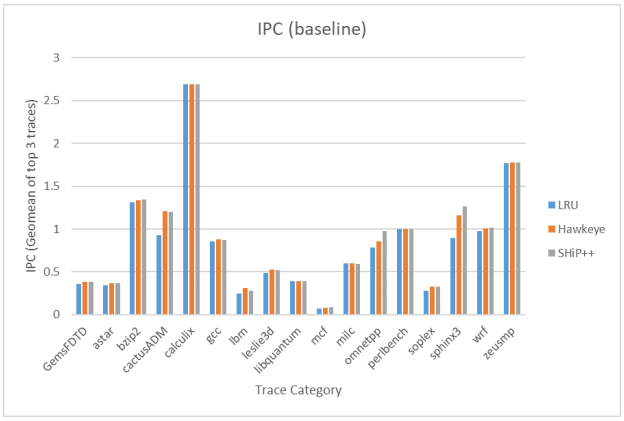

Fig. 3. IPC for different trace Categories

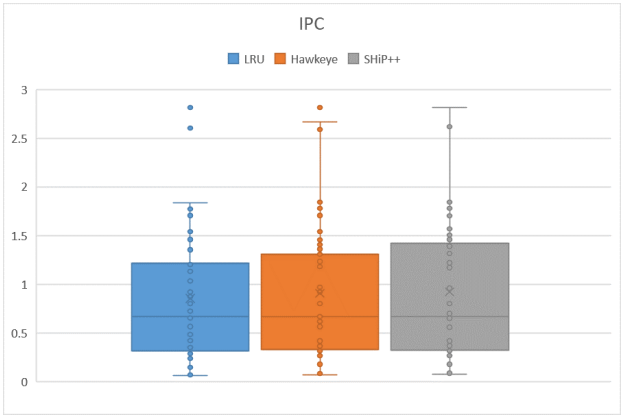

Fig. 4. IPC for different trace Categories

We can see from Fig. 1 that LRU give a better (lower) MPKI in quiet a few trace categories (ex: GemsFDTD). But despite the lower MPKI the actual performance of hawkeye and SHiP++ is better (higher IPC) as seen in Fig. 3. This indicates that MPKI is only a crude indicator of the performance.

IV. Design space exploration

A. Hawkeye

1) Number of OPTgen entries: ”Occupancy vectors (OPTgen) measure time in terms of cache accesses to the corresponding set, and they include enough entries to model 8× the size of the set (or the associativity). Thus, for a 16-way set-associative cache, each occupancy vector has 128 entries (corresponding to 8× the capacity of the set), and each occupancy vector entry is 4 bits wide, as its value cannot exceed 16.”

The above statement is a quote from ”Back to the Future: Leveraging Belady’s Algorithm for Improved Cache Replacement” which is the backbone of the Hawkeye design. To verify the correctness of this quote and if it is still applicable to the modification made by the Hawkeye design, the number of OPTgen entries are varied and performance is reocrded.

OPTGEN Vector Size	MPKI	IPC
32	19.97330608	0.654321344
64	18.85032843	0.664816844
128	18.89216725	0.66416129
256	18.90550176	0.663606609
512	18.91470804	0.66346778

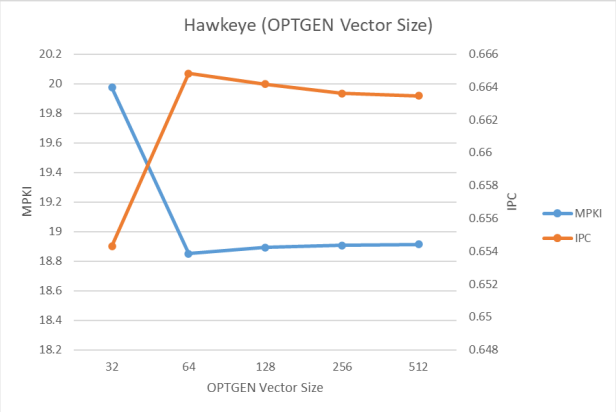

Fig. 5. OPTGEN VECTOR SIZE vs MPKI and IPC

We can see form the results that lowering of OPTgen entries decreases performance but increasing it does not improve performance. This observation (crudely) verifies the above statement. The reason we see the drop in performance between 32 and 64 size (instaed of 64 and 128) could be due to the use of two predictors that distinguish between demand accesses and prefetch accesses. This could reduce the required number of entries from 128 to 64.

2) Number of ways in Sampler structure: The Sampler is a structure that keeps track of metadata and uses PC to index into the tables. Changing the number of way in this data structure directly affects performance as it may lead to more or less aliasing.

Sampler ways	MPKI	IPC
2	18.98267569	0.660166631
4	18.90461765	0.663176363
8	18.89216725	0.66416129
16	18.85851843	0.664711661

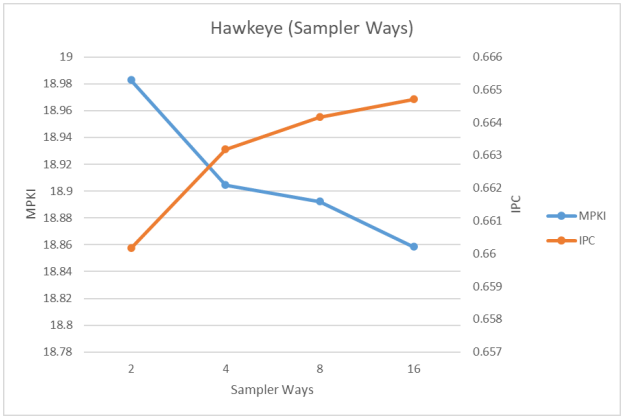

Fig. 6. Number of ways in Sampler vs MPKI and IPC

Fig. 6 shows that higher associativity (ways) leads to better performance. This is because of lowered aliasing.

3) Maximum value of RRIP: The RRIP stores the priority of the cache line. When a cache line needs to be evicted the one with the lowest priority is chosen. Changing the maximum allowed value of RRIP would have a direct impact the granularity with which priority is stored and thus choice which line is being chosen for eviction. This would directly affect performance.

maxRRPV	MPKI	IPC
3	18.86467804	0.665833441
5	18.87877824	0.664790967
7	18.89216725	0.66416129
9	18.89641667	0.663886043
11	18.9036451	0.663525416
15	18.93779647	0.662764187

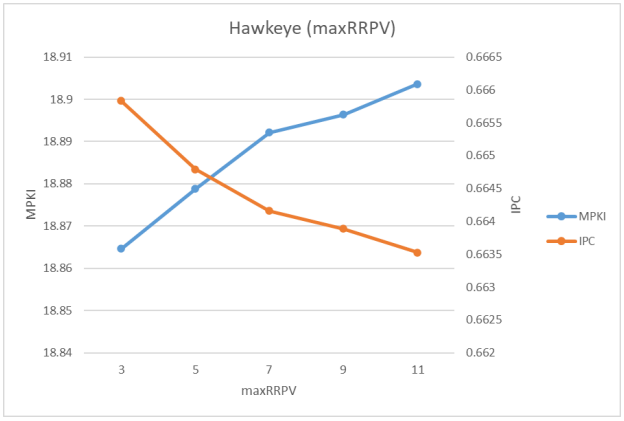

Fig. 7. Maximum value of RRIP vs MPKI and IPC

Contrary to normal expectation, lower maximum value of RRIP is giving better performance. This could be due to the memory access behaviour of the chosen traces.

B. SHiP++

1) Maximum value of SHCT counter: SHCT counter tracks the re-reference of accesses. The max value of this counter determines how long it takes to train the SHCT to insert a cache line with RRIP value of 0 or 3. Changing the maximum allowed value has a direct impact on the performance.

Max. SHCTR	MPKI	IPC
3	19.41578902	0.66683417
5	19.29676804	0.666952681
7	18.88401843	0.668983423
9	19.14527431	0.667101038
11	19.12901196	0.667758434

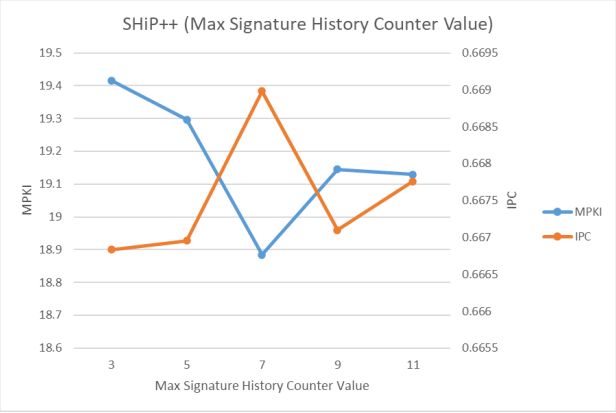

Fig. 8. Maximum value of SHCT counter vs MPKI and IPC

Fig. 8 shows that the max value of 7 yield the best performance. If the max value is too low then, a access would become useful too soon and if this value to too high then it would take much longer for re-refrences to reach the status of useful.

2) Signature length (SHCT Size): The lsb of the PC is used as signature to index into the SHCT. A longer signature would reduce aliasing, which would directly impact performance. A longer signature would also require more entries in the SHCT.

SHCT Entries	MPKI	IPC
4K	18.9345051	0.668353946
8K	18.9053802	0.66851043
16K	18.88401843	0.668983423
32K	18.87242804	0.669502274
64K	18.86316529	0.669463617

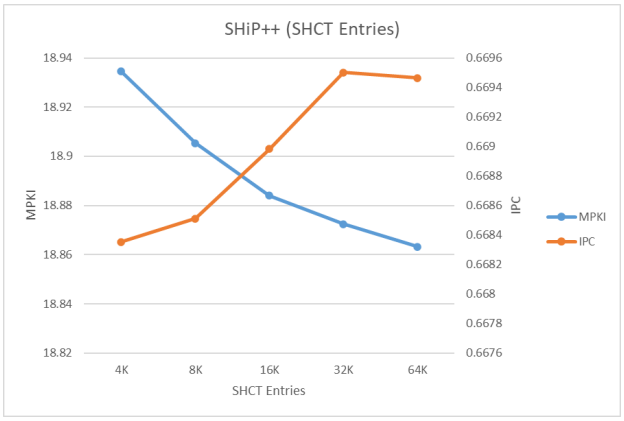

Fig. 9. SHCT SIZE vs MPKI and IPC

We can see a steady increase (improvement) in performance as the size of the SHCT is increased. This is because of the reduced aliasing in the SHCT.

3) Number of Leader sets: To reduce the hardware size the tracking is done only on a subset of the sets in the cache. The SHiP++ paper refers to this as the Leader sets. Changing the number of leader sets should affect the performance as it changes the set which are being tracked.

Number of Leader Sets	MPKI	IPC
8	18.82134745	0.66579706
16	18.86895961	0.668018434
32	18.89923863	0.66767409
64	18.88401843	0.668983423
128	18.96860216	0.668475207
256	19.02277	0.669065579
512	19.18468902	0.667938779

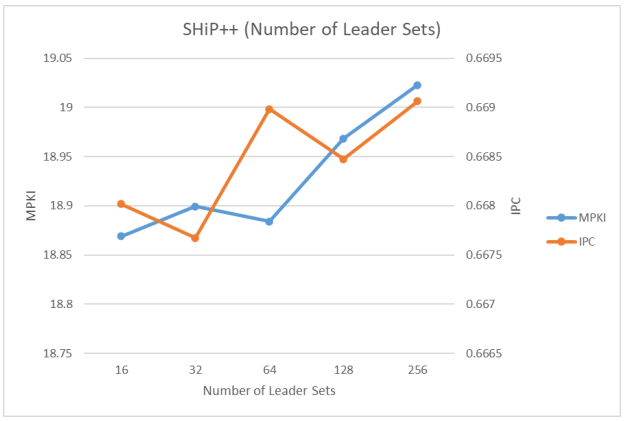

Fig. 10. Number of Leader Sets vs MPKI and IPC

Fig. 10 shows that lower number of Leader Sets give worse performance. As the leader sets increase the performance improves. The performance at 128 is an anomaly probably because it causes more aliasing in the SCHT as it size is not being scaled with number of leader sets.

V. Hardware budget exploration

For exploring the hardware budget different size constraints are used on the two policies, Hawkeye and SHiP++ and the performance is recorded. For Hawkeye the following data strcutures were changed to reduce and increase the hardware budget:

- RRIP counters (maxRRPV)
- Hawkeye Predictors size (SHCT_SIZE_BITS)
- OPTgen vector size (OPTGEN_VECTOR_SIZE)
- Number of entries in Sampler cache (SAMPLED_CACHE_SIZE)

For SHiP++ the following data strcutures were changed to reduce and increase the hardware budget:

- RRIP counters (maxRRPV)
- Number of Leader Sets (NUM_LEADER_SETS)
- Number of SHCT entries (SHCT_SIZE)

Size (KiB)	16	32	64
Hawkeye (IPC)	0.6566825	0.6641612	0.6634734
Hawkeye (MPKI)	19.401341	18.892167	18.900734
SHiP++ (IPC)	0.6689834	0.6644259	0.6616791
SHiP++ (MPKI)	18.884018	19.441335	19.679205

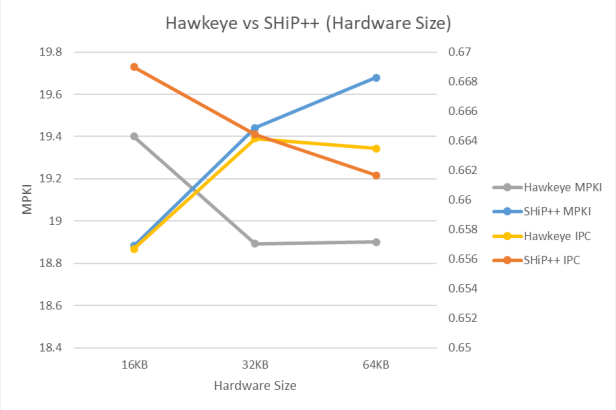

Fig. 11. MPKI and IPC of Hawkeye and SHiP++ of different sizes

The size used by Hawkeye paper for CRC2 was 31.8KB while SHiP only used 16KB size even though the allowed max size was 32KB.We can see the reason behind it on the Fig. 11. The performance actually decreases as the size increases for SHiP++. For Hawkeye we see worse performance for smaller (16KB) size (as expected) but for 64KB size the performance remains the same.

VI. Hardware budget constraint

The size of LLC in CRC2 simulator is 2MB. The size of the replacement policies were of the order 16KB to 32KB.

VII. L1 I-Cache vs LLC

Comparing this to the size allowed for the replacement policy (32KB) the overhead is about 1.6%. For the same overhead if the L1 cache (32KB) is designed to have a similar structure it's size would come out to be 512 bytes which is too small to do anything with.

For L1 I-cache prefetchers the size is of the order 128KB which is 4 times the size of the L1 I-cache itself. This makes it a 400% overhead. The reason these overheads are acceptable is that L1 miss penalty is very high and so it is desirable to spend more resources to increase L1 hit rate. Also prefetcher is is not on the critical path so it can be a bit big and slow as long as it is able to timely prefetch with high accuracy.

VIII. Constraints on Latency

For L1 I-cache prefetchers the size is of the order 128KB which is 4 times the size of the L1 I-cache itself. This makes it a 400% overhead. The reason these overheads are acceptable is that L1 miss penalty per unit time is very high (because of the high number of L1 accesses) and so it is desirable to spend more resources to increase L1 hit rate. Also prefetcher is is not on the critical path so it can be a bit big and slow as long as it is able to timely prefetch with high accuracy.

Replacement policy is different in this aspect, it is on the critical path. When a cache miss occurs the replacement hardware has to evict an old cache line and place the new cache line. If it takes too long to do this then the miss penalty increases which is not acceptable.

LLC itself is very big. A 400% overhead would be too big to fit on chip. LLC already occupies most of the area in many chips (processors).

- SHiP++ - Updating RRIP values when no max value is found for eviction
- SHiP++ - Indexing into the very big SHCT
- Hawkeye - accessing big structures like OPTgen, predictor, and RRIP values sequentially

These structures or computations cannot be present on the front end of the processor because the front end needs to be fast to have god performance. It should be able to issue multiple instructions every cycle or the performance (IPC) will drop.

IX. Infrastructure Development

The code can be downloaded from github: https://github.com/tapatil/CSE240C_SA2

Use the my_run.sh shell script to build and run the code.

A. Setup

The simulator used is the latest version of ChampSim (dated 2/9/2022). The .cc files of Hawkeye and SHiP++ are downloaded from the CRC2 website. A new file is created everytime with updated macros to run experiments.

B. Changes Required

The CRC2 was done on an old version of ChampSim. The New version made few changes to the way the compilation is done and the interface of the Replacement policy functions interfaces. So the Original program of Hawkeye and SHiP++ had to be edited to make them compatible with the new version

of ChampSim. All the changes made are summarized in the ”my_run.sh” script file.

C. Scripts

The script (”my_run.sh”) uses ”llc_replacement_policy” array and ”allTraces” array. Uncomment the required LLC replacement policies from ”llc_replacement_policy” array and run the script. The traces are run in parallel threads. The number of threads can be changed using the “numThreads” variable. The script uses a C program to calculate the arithmetic and geometric means. This is done so that precision is not lost during calculations.
Refer to that last few line for control what the script does. You may comment out the parts that are not required. The ”compile_if” function compiles the files specified by ”llc_replacement_policy” array if they are not already compiled. The ”run” is used to run all the files specified by ”llc_replacement_policy” array on all the traces specified by ”allTraces” array. The ”get_IPC” and ”get_MPKI_LLC” compiles the results from the raw files generated from the ”run” function.
To make the compilation seamless, and new folder and file named ”UUT” (unit under test) is used. The specified llc replacement policy file is copied to this ”UUT.cc” file and then compiled. With this setup the ”champsim_config.json” does not need to be edited every time.

D. Output

The script collects all the data in text files. The raw output is collected in ”results_raw folder”. The ”results_ipc” and ”results_LLC_miss” contains one file per replacement policy that has the IPCs and LLC Misses from all the traces. The ”results_ipc_mean” and ”results_LLC_mpki_mean” also contains one file per replacement policy that has the arithmetic and geometric means of IPCs and LLC Misses.