# CSE-240C Simulation Assignment 2

Gandhar Deshpande

## ABSTRACT

This report is written in fulfillment of Simulation Assignment 2 for the course CSE240C Advanced Microarchitecture, Winter 2022 at the University of California, San Diego. It looks at the submissions for the second Cache Replacement Championship. It gives a overview of three of the submissions at a high level. It analyzes two of them in detail and reproduces their results. It then looks at two major design factors in the designs and tries to run an design space exploration to understand the impact of these factors on performance. Finally it looks at hardware budgets, and how the designs scale up and down with the budget. It compares the design of a front-end component like a Prefetcher and a memory subcomponent like the replacement policy in terms of area and latency.

## 1 Introduction

Every processor in the today's world has 3 levels of cache between the processor and the main memory. While typically the first level is private for every core, and the second one may be private or shared, the third level is typically always shared. These grow progressively in size as we go further away from the CPU towards the memory. Since the size is limited, it stands to reason that at some point the cache will be full and will have to replace some values.

To support this, every processor will have a replacement policy for the cache. Some of the more common and simple policies are First-In-First-Out policy, a random policy or a Least-Recently-Used policy. However, since the third level of cache is shared and usually stores a large amount of data, it makes sense to make sure that the replacement policy here is as accurate, to reduce the penalty of a cache miss substantially. Along with this, recent processors also have prefetchers built into caches to prefetch values based on instructions, so that the values are available when the request for data actually comes from the processor.

In this assignment, we look at some of the proposals for replacement policies which are intended to provide a better performance in the processors. Some of these policies extend existing policies, while some others propose entirely new ideas, leveraging the addition of prefetchers. We analyze the designs and try to understand what impact a better replacement policy could make on a processor.

The report is arranged as follows:

- Section 2 discusses three of the winners in the competition at a high level.
- Section 3 discusses the two of the replacement policies in detail going over various design factors.
- Section 4 shows the results obtained by the prefetchers.
- Section 5 analyzes some of the factors in the design which would affect the performance and shows a design space exploration over those factors.
- Section 6 analyzes the trend shown in this design space exploration
- Section 7 looks at the scalability of the design and produces the results at a lesser hardware budget
- Section 8 finds the trends in scalability.
- Section 9 looks at the constraints in area in a replacement policy versus the constraints in a instruction prefetcher.
- Section 10 looks at the constraints in latency and computation in a replacement policy.

## 2 Three winners

The three winners that I chose to analyze here are **SHIP++**, **Less Is More(LIME)** and **Reuse Detection(ReD)**. These three designs try different approaches to solving the problem of cache replacement.

**SHIP++** is a design that builds up on the preexisting algorithm SHIP by enhancing it with information which is known for every processor. The policy targets to make SHIP cache-access aware, and modify training and predictions. It provides 5 different enhancements over the existing model which allow a better performance. It changes the way cache insertions are handled, using information It uses the existing data structures of SHIP, and adds a few more to implement these features.

**Less Is More (LIME)** tries to take the optimal predictions on demand based accesses using Belady's algorithm for replacement. It uses past information make predictions about whether a particular cache access should insert into cache or not based on past access information. It samples 20 sets and employs Belady's trainers to make predictions based on the behaviours of these sampled sets. It creates different bins based on the behaviour, so that an instruction once classified does not need to be trained by the algorithm again, and can simply refer to the previous behaviour. Finally since evictions cannot be avoided and writebacks are not allowed to be bypassed, it uses SRRIP algorithm to decide victim cache lines.

**ReD** works on the idea that we only want to store data which has demonstrated reuse. For this, they track addresses that miss in the LLC, and also the PCs that request those addresses. Address Reuse Table(ART) stores the address which miss in LLC, a miss in LLC and ART is candidate for bypassing to main memory and added to ART. A miss in LLC but hit in ART is stored in LLC, as it has demonstrated reuse. This means that any request would have to miss twice to be considered a candidate for caching. To avoid this, the paper also uses a PC reuse table that is indexed by the PC requesting the address and keeps a reuse and noreuse counter. If the PCRT says that reuse value is high, the bypass mark is ignored and data is stored. The minimum reuse probability is set to 1/4 indicating a ratio of reuse to no reuse of 3. It also uses SRRIP to keep track of actual evictions.

## 3 Design of winners

In this section, we look at the details of two of the winners discussed in the previous section - SHIP++ and LIME.

### 3.1 SHIP++

SHIP++ is a replacement policy built on top of the SHIP policy, meant to enhance it to provide a better performance with awareness of access type. It adds 5 new optimizations based on type of cache access and modify the training and the update policy to reflect specific values for the LLC. I'll briefly go over each of the enhancements below

**Improved Cache Insertion**

Baseline SHIP installs all lines with a value of 2, and attempts to learn the behaviour of re-referencing. When SHIP notices that a particular entry in the SHCT is saturated at 0, it inferences that insertions made by this PC are rarely referenced again. So all new lines referenced by this PC are added with a RRPV value of 3(highest value). Similarly, if a counter is saturated to max value, future insertions for this PC are installed with RRPV 0 since the relearning process is not necessary.

**Improved SHCT training**

SHiP trains the SHCT table on cache hits and cache evictions. However, training on cache hits disproportionately trains the hits against the misses, while allowing for quick training. This policy chooses to train only on the first re-reference to make sure that the training is done proportionate to the hits and misses.

**Writeback-aware RRPV Updates**

Since writeback updates are non-demand background requests, all such cache lines are installed with a static RRPV value of 3. This signifies that these are low priority requests and do not occupy the cache for a long period of time.

**Prefetch-Aware SHCT Training**

Prefetch and demand accesses might have vastly different access patterns. Since we are using the same table for training for both prefetch and demand accesses, this might end up reducing the performance. To counter this, a is_prefetch flag is maintained which is used to track if the request is prefetched and requests are indexed using PC appended with the prefetch bit.

**Prefetch-Aware RRPV Updates**

SHIP statically updates RRPV to 0 whenever there is a hit. There are two scenarios where the RRPV update changes based on type of access. In first scenario, if a demand access hits on a prefetched line, the RRPV is set to 3, since it is observed that such lines are prefetched for this explicit demand and will likely never be used again. To signify their low usage after, it is given a RRPV value of 3. Note that this is only for the first demand access after the prefetch, any subsequent accesses would set the RRPV to 0 as expected in the baseline. The second scenario, where a prefetched line is re-referenced by a prefetcher access, the RRPV is not updated to 0, but kept as is.

### 3.2 LIME

The LIME replacement policy uses 3 major components - Belady Trainers, PC Classifiers and SRRIP counters. These three work together to ensure that the policy is implemented and provides better results compared to the baseline LRU.

The Belady trainers are employed on 20 sampled sets from the cache. Each trainer observes accesses and relates them with a history of accesses to decide whether a particular access is cache friendly or not. For this, it uses a history of 16 times the associativity of the cache, in this case, 128 entries implemented as a FIFO queue. Each entry consists of a PC, a data address, an occupancy vector and a hit flag. The PC is kept to correlate the the data access to the address from a previous usage. For budget constraints, they keep 18 hashed bits for PC and 37 hashed bits of data address. The occupancy vector is used to keep track of all the ways that are filled with useful, friendly data. The hit flag is enabled when the Belady trainer decides to cache the data. When an entry is being pushed out of the FIFO, it goes into one of the bins of the PC classifier based on the value of the HIT flag.

PC classifiers classify in three bins - KEEP, BYPASS and RANDOM. It defines three different instruction states - KEEP, BYPASS and NA. All instructions which are not trained are in NA state. Once trained, instructions go into KEEP or BYPASS bin. It may be possible that a PC shows one access pattern during a phase of execution and another pattern during another phase. In such a case, these go into random bin. KEEP adn BYPASS are implemented as a Bloom filter. The disadvantage of Bloom filters is that data cannot be removed, which is why non-deterministic values go into RANDOM. RANDOM bin is implemented as a LUT. All bins are searched in parallel to get a hit. If RANDOM, the classification there is used, else KEEP or BYPASS is used. If PC is not found in either bins, it belongs in NA categoty(Never trained by Belady trainer)

The evictions and updates are done using a 3 bit Re-Reference Prediction Value counter, which tracks the possibility of being re-referenced. When replacing, each way's counter is checked and the one that matches the max value is replaced. If none are found, all counters are incremented and searched again. This continues till the first occurrence of the max RRPV value. Normally any new line installed has a counter value of 0, but in this case, they change the update policy a bit to reflect the cateogories. NA on a cache hit is installed with 3, and on a miss is installed with 6. KEEP is installed with a 0 on a hit and with 0 and aging all other counters on a miss. BYPASS is the same as NA, showing that an unclassified PC is given an amount of BYPASS, unless proven otherwise.

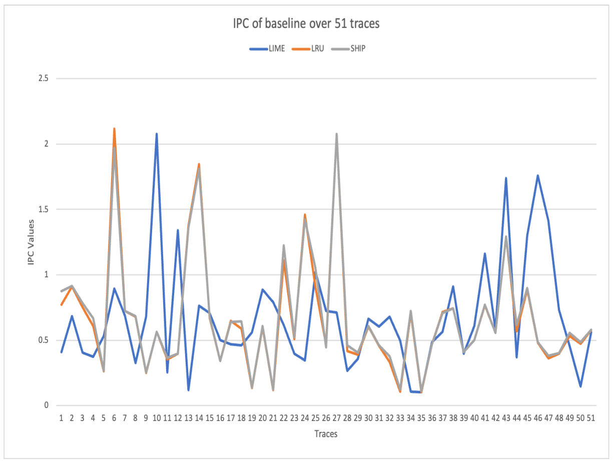

**Figure 1: Performance of Winner Replacement Policies**

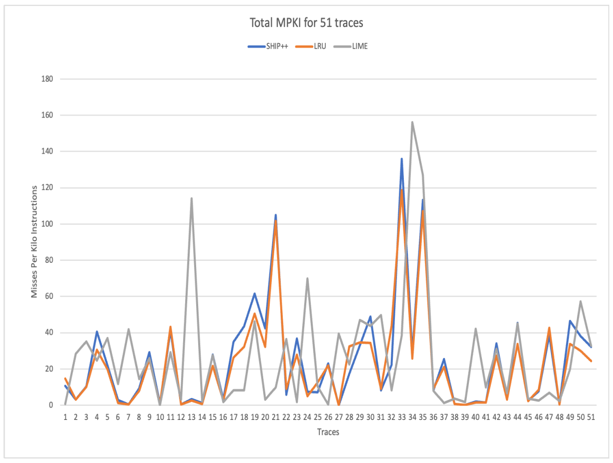

**Figure 2: MPKI of Winner Replacement Policies**

To work with the Prefetch and Writeback accesses, LIME does not consider them during training, since writebacks are written back and prefetches have a large possibility of not being used due to not very high accuracy rate. However, they do keep in mind that these have some level of accuracy, which means that they should not be ignored altogether, so while they are not trained, they are kept in a selected way in the cache(way 0 is chosen by design).

## 4 Reproducing the winner results

In this section, I will plot the results of the baseline of SHIP++ and LIME against the baseline LRU model. All the policies will have prefetchers configured. We use a Next line prefetcher for L1 and IP based stride prefetcher for L2. For evaluation, I use a mix of 17 sets with a total of 51 traces. For the baseline I plot two figures, the IPC performance of the three policies over the 51 traces and the MPKI for the 51 traces. These 17 sets include sets like astar, bzip2, cactusADM, calculix, gcc, GemsFDTD, lbm, leslie3d, libquantum, mcf, milc, omnetpp, perlbench, soplex, sphinx3, wrf and zeusmp.

When compared to the reported statistics in the papers, I find that we gain an IPC Speedup of 2.7% for SHIP++ which is a bit different from what the paper reports. For LIME, I get a IPC speedup of 1.4% which is much less as compared to the paper reporting. However, the paper has used a different set of traces for evaluation, and does not mention the number of instructions simulated. This may be a major reason for this difference in numbers, since even a few good performing traces can improve the IPC drastically as seen from the image above. We see from the graphs that SHIP++ follows similar performance to LRU, while LIME performs well on different workloads.

## 5 Design Space Exploration

In terms of design space exploration, I chose to vary parameters which would not affect the size budget very much. I tried to vary each parameter to the highest level possible to gain some insight on how the performance would vary with different workloads. I used the set of traces used in the SHIP++ paper. All traces are simulated with 10 million instructions for warmup and 100 million for evaluation, which is the criteria used by SHIP++. LIME does not mention how many instructions they use for simulation, but they use a mix of traces, a large part of which are common with this set.

For SHIP++, the parameters I chose were the number of sampled sets used. The SHIP++ model uses 64 sets. I varied the number from 16 to 128, to see variations in performance and MPKI. The other parameter I chose was the maximum value of the RRPV, which is used in the algorithm to decide which way to evict. The paper uses 3 as the highest value. I varied it from 2 to 7 to see how the performance varies as evictions become slower.

For LIME, I varied the hashing used for PC and Tag. The paper mentions these values as 18 and 37 to be reached empirically and for budget reasons, so i decided to run experiments to try to verify the performance around these values. I use the same hashing mechanism - take n-2 lower bits and append the 32nd and the 40th bit for PC and take the 8 highest bits of the tag and append to lower N-8 bits for tag. I varied the PC from 14 to 22 bits and the tag from 34 to 40.

## 6 Evaluation and Analysis

Since all designs exploration is conducted on 51 traces which are 3 traces per set and 17 such sets, all comparisons are made against a baseline LRU implementation with the same traces. I also use the same number of warmup and simulation instructions for evaluation across the entire design space exploration.

Figure 3 shows the IPC for different values of RRPV. We can see that the IPC is highest at a value of 3. It is pretty close at a value of 2, but quickly starts losing in performance as we starting increasing the value. Since a lot of enhancements rely on this Max value of RRPV, it may be that unnecessary block of cache are now taking longer time to be evicted, resulting in the lower performance. Figure 4 shows the MPKI increasing as we increase the RRPV, which further proves the choice of 3 as the max value.

Figure 5 shows the variation in IPC as I change the number of sampled sets used in SHIP++. As I move the number from 16 to 128 in increments of 16, we see that the IPC is varying a bit, with no particular trend. However, it is necessary to

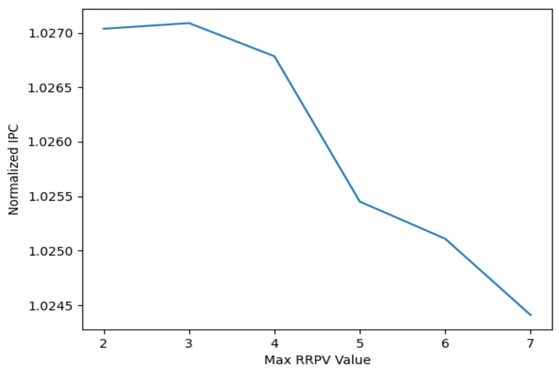

**Figure 3: IPC for RRPV high value variations in SHIP++**

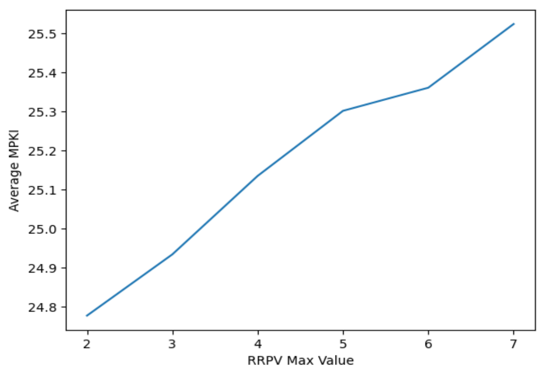

**Figure 4: MPKI for RRPV high value variations in SHIP++**

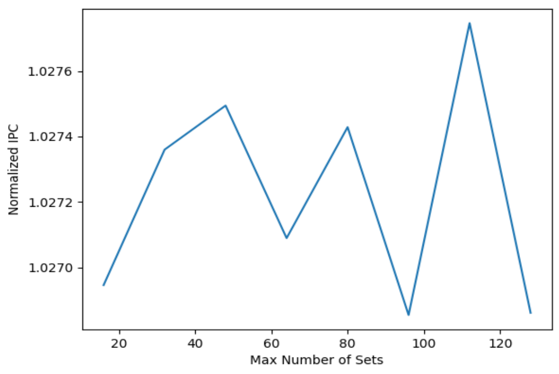

**Figure 5: Variations in the number of sets in SHIP++**

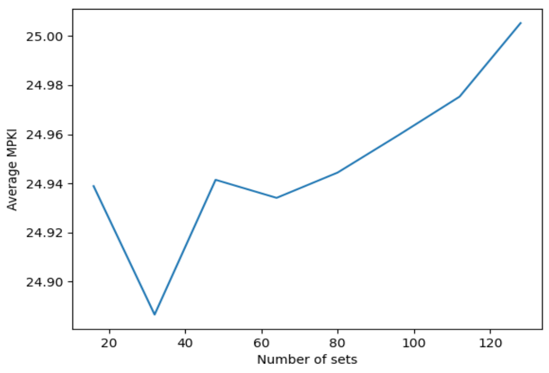

**Figure 6: MPKI for variations in number of sets in SHIP++**

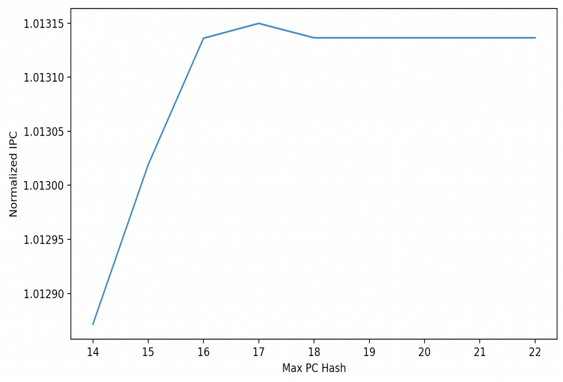

**Figure 7: IPC for Variations in PC Hash size in LIME**

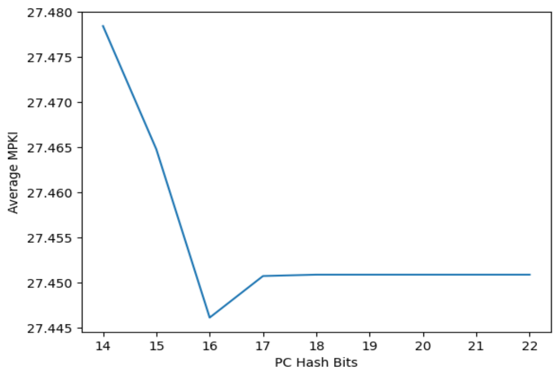

**Figure 8: MPKI for Variations in PC Hash size in LIME**

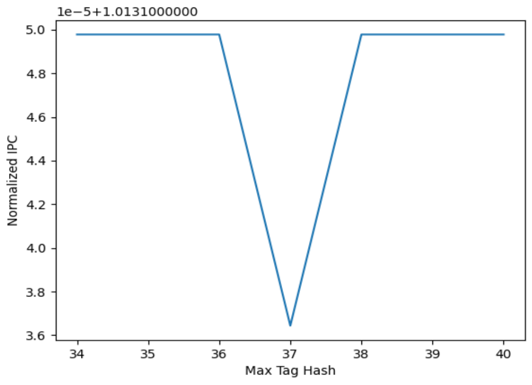

**Figure 9: IPC for Variations in Tag Hash size in LIME**

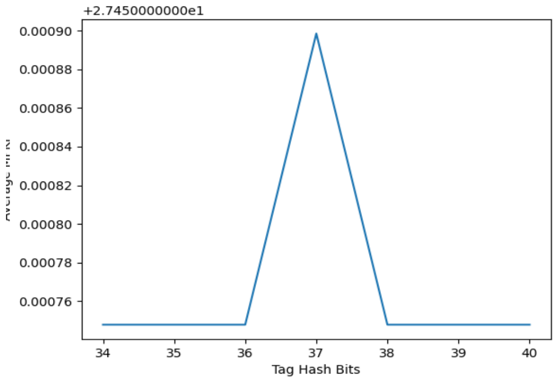

**Figure 10: MPKI for Variations in Tag Hash size in LIME**

point out that the variation here is only in the fourth decimal place, so the performance is not really being affected much. I believe 64 is chosen as an adequate number which might be necessary while working with multicore configurations. It might be better to perhaps use 48 sets here which would bring optimal performance at a budget. We can see that 112 sets show the best performance, but 112 sets also means a larger increase in the budget, which might have prompted a decision to choose 64 sets at a slightly lower performance. Figure 6 shows the MPKI which is lowest for 32, and higher for 48 sets. This might have been another factor while choosing 64 sets in the paper.

Figure 7 shows the IPC results for variation of the number of hashed bits from PC being used for LIME. Since the PC is used to classify the the addresses into different classifiers, we can see that a variation in PC from 12 to 16 is in a linear increasing fashion. At 17, the value chosen by the paper, the IPC is optimal, and it slightly dips and stays constant after that for all sizes. This leads to the implication that 17 bits are optimally enough to classify all PCs for the given workloads, but anything above 17 would in effect bring just about an equal performance. We see that MPKI is in a lowest for 16 bits, but is not much higher for 17 bits, and provides a better IPC at 17 bits.

Figure 9 shows the IPC distribution for different hash sizes of the address tag. From this design, we can conclude that 37 bits is more than sufficient to compare the addresses, in fact even 34 bits would be just about the same. Once again, it is necessary to point out that all the variations shown here are at the fifth decimal place, so in effect they do not really affect teh performance much. It might be a good idea to explore by reducing the tag bits to a much lesser level to find out at what size the performance starts deteriorating. Figure 10 shows a similar figure in terms of MPKI, again changing only at one point in an inverse of the IPC, varying only on the 5th decimal place.

## 7 Scalability

To understand how the policies would scale, we first need to consider what data structures in each policy take up a large space. We also need to consider when varying size, what data structure would dominate to the extent that all others are relatively very small.

For SHIP++, we have data structures like the RRPV table, the is_prefetched table, the Signature History Counter Table, the Sampled Sets and the Sampled Set IDs. Of these, the Sampled sets were varied in DSE and they increase linearly in relatively small steps. The RRPV table increases size by 4 KB for every extra bit, but a 3-bit counter would become more than sufficient and any more could possibly be counter productive. However, the signature table depends on the signature length - a 14 bit signature uses a 16K entry table. For every extra bit in the signature, the table size would double, which means at higher budget, this data structure would dominate heavily. In my experiment, I varied the size from 12 to 18, making the size of the table from 1.5KB to 96KB. Figure shows the performance across different sizes.

For LIME, following a similar train of thought, I looked at the various different data structures. In this, I find that number of trainers and the history length are both factors which would influence the size. To check multiple combinations, I ran simulations with 5, 20 and 40 trainers as well as history lengths of 64, 128 and 256 in different permutations and combinations. Some of them end up with same size but could very well perform differently.

## 8 Scalability Evaluation and Analysis

From the figures shown, we can see how the performance varies as we increase the first order structures which largely influence the size of the policy.

We now try to examine the reasons as to why the performance of the two policies at scale might be as it is showing.

For SHIP++, I varied the number of its used in signature of the PC used for indexing the Signature Hit Counter Table. Since this is an indexing mechanism, each extra bit results in a doubling the size of the SHCT. To compare the values I use signature of 12 bits to up to 18 bits. From the graph we see that the performance increases from around 12 bits to 15 bits of signature after which it drops down. The reason for increase is simple, we are getting more information which is useful and is being used, improving performance. We see from the MPKI figure that it is indeed moving in an inverse manner to the IPC, showing that the the choice of 15 bits would be most optimal in terms of performance, but may increase the budget a bit.

However, we see a dip in IPC after 15 bits, which doesn't follow the logic from before. Here what might help is understanding the fact that the number of sampled sets plays an important role, and as such, if we keep them to a constant value, at 16 bits of signature we are not getting enough information out of the sampled sets while having a lot more counters. I believe if the number of sets is scaled up along with this, it might remedy the situation.

For LIME, we see the graph which shows variations for the number of trainers and the history length. I chose one value each above and below the one selected in the paper to show contrasts at low and higher budget. We see that at almost all budgets, 20 trainers seem to be enough to train, and having a larger number of trainers is not providing much of a performance boost, or is stabilizing to the same value. We do see variation in the size of history. As we increase the history to only 64 bits, we see a lesser performance compared to the 128 history. Surprisingly, we also see reduced performance for 256 entries in the history. While the performance increases for 5 trainers as we increase the history, it does not follow the same trend for 20 and 40 trainers, leading to a conclusion that 128 history bits are enough to classify the addresses and increasing history does not help, but hinders. I believe that might be because the address is not put into classifier until it reaches the head of the history FIFO queue, which takes a longer time as we increase the history length.

From Figure 14, we see the MPKI for different number of trainers and and history lengths and we see that the misses reduce with reducing number of trainers and increasing history length. This once again proves the choice of 128 entries of history to be a tradeoff between IPC and reduced MPKI. We can judge that this would not scale well with size.

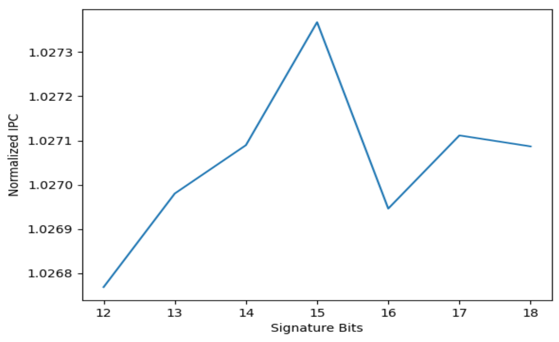

**Figure 11: IPC for varying the number of bits used in signature**

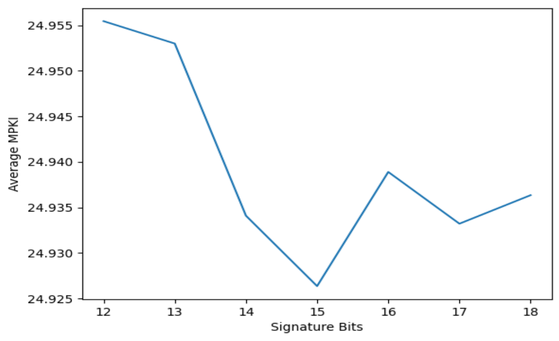

**Figure 12: MPKI for Varying the number of bits used in signature**

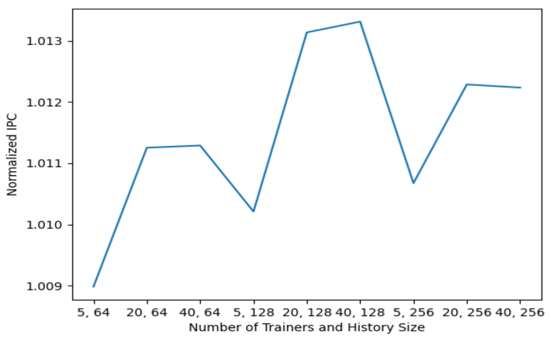

**Figure 13: IPC for Varying number of trainers and the history length**

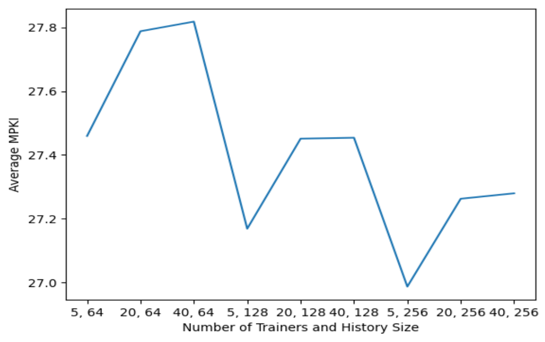

**Figure 14: MPKI for Varying number of trainers and the history length**

## 9 Area Constraints

This section looks at the area differences between a prefetcher and a replacement policy and how much overheads they use over the structure they assist.

To begin the comparison, the replacement policy is used for the Last Level Cache, which is 2MB in size, while the instruction cache used in the prefetcher is 32KB. In the replacement policies that we look at, the hardware budget required for these policies is a maximum of 32KB, but most use a little less than that. SHIP++ utilizes only 20KB per core when prefetchers are used, while LIME utilizes about 31.2 KB per core. This turns out to around 1.56% of the overall hardware budget for the replacement policies.

Comparing the same for the prefetchers, we see that the instruction cache for L1 is 32KB, while the prefetcher is allowed a budget of 128KB for this cache. The policies examined in previous work use around 110KB for FNL+MMA and around 120KB for BARCA. This turns out to be nearly 400% of the budget for the cache itself.

When comparing this with the LLC, it makes one wonder why would one allocate so much budget for a smaller data structure while allotting such a small budget for the larger LLC cache. One reason might be that LLC cache is very far from the processor, so performance improvement might not be that great. Also the existing policies work pretty well and new models do not provide that high an improvement. On the other hand, prefetchers are relatively recent and have a larger scope for predicting which values to prefetch, which creates more variations and allows for better performance. We see this also from the performance improvement seen in the prefetcher versus the LLC where we also see the size variations.

## 10 Latency Constraints

When comparing the latency in terms of computations required, the LLC replacement policy does not have a large number of complex computations and the prefetchers usually employ more complicated logic for their computations. With that said, I will try to list here computations that might turn out to be slower compared to the prefetcher.

The first computation is of selecting the way to evict, which is done in most algorithms by using a RRPV. The algorithm is iterative and chooses the first way that has the maximum value for the RRPV. In case of no such value, all RRPV counters are incremented and the search begins again. Such a computation would likely result in slower execution, leading to non-timely prefetches.

Another is that the prefetcher is always running, not dependent on any signal per se to run. Of course there is some dependency on branch prediction, but even then, the results can be chosen speculatively and prefetching can begin. Compared to this, the LLC policy acts only on memory access requests, whether they are demand based, write backs or prefetches. This may result in some dependencies which lead to a slower execution, which cannot be afforded in a prefetcher.

Lastly, since the LLC is so big, any replacement policy access to a set would take a longer time due to the slower implementation of the microarchitecture. This is not the case in prefetcher which works on a relatively smaller sized cache and works at a much higher speed, prefetching multiple results at every cycle.

## 11 Implementation details

The code used and implemented in this project can be found in the public github repository : `https://github.com/gdpande97/CSE240C` inside folder Sim2. The repository also contains all the result files generated from the various simulations. It also contains all the scripts written to run the simulations in a batch mode and script to extract relevant values from the results. Along with this, it also contains the code of the simulator and a Steps-to-run.txt which details how the simulations were run in the environment. All the relevant files are added to the repository.

## 12 Acknowledgments

The whole study is done using the infrastructure provided by the 1st Instruction Prefetching Championship. I use their simulator for simulating all the prefetchers, with the logic for all mechanisms apart from the prefetcher provided by the Championship. I also use the code made available by the various winners in the competition to the competition to simulate and explore the design space of the two designs studied in this paper.

## 13 References

[1] Vinson Young, Aamer Jaleel, Chia-Chen Chou, Moinuddin Qureshi. SHiP++: Enhancing Signature-Based Hit Predictor for Improved Cache Performance

[2] Jiajun Wang, Lu Zhang, Reena Panda and Lizy Kurian John. Less is More: Leveraging Belady's Algorithm with Demand-based Learning

[3] Daniel A. Jimenez, Paul V. Gratz, Gino Chacon, Nathan Gober BARCA: Branch Agnostic Region Searching Algorithm.

[4] Andre Seznec. The FNL+MMA Instruction Cache Prefetcher.