Exploring Adaptive Cache Management for Instruction Data

Gandhar Deshpande

ABSTRACT

This report summarizes the simulations workloads done in partial fulfillment of the course CSE240C - Advanced Microarchitecture. We look at management of cache components like Replacement policies, and Instruction Prefetchers. We look at the different designs, select 2 of them to explore in detail and give our insights about the designs. We finally combine the two replacement policies to work together in an adaptive manner following a concept called set dueling to improve the performance of the overall replacement policies to be better than either of the replacement policies.

1 Introduction

Memory is one of the fundamental components of any processor. Over time, as Dennard scaling and Moore's Law pushed the processors to be smaller and faster, the memory did not scale in the same manner. This gap has been increasing as we moved to multicore machines for performance. To mitigate the performance degradation due to this gap, computer architects have used various methods. Largely these work around the cache memory, which allows fast access to memory for the processors.

Architects have added multiple components to make sure that the data in cache stays relevant. Some of these components are a multi-level cache hierarchy, a cache replacement policy and prefetchers. These components reduce the bottleneck significantly, allowing it to function at a much higher rate. A cache hierarchy allows us to use different levels of memory with a smaller size but a faster access rate, reducing the latency in the execution of the processor. However, these are only useful, if we can keep the relevant information in them, otherwise the request ends up going to the main memory.

To counter this, architects came up with a replacement policy that will remove the useless data and keep only useful data in the cache. There are many different policies that do this, ranging from simple algorithms like FIFO and Random, to more complex ones. Apart from this, typically caches are divided into instruction and data caches. For instruction caches, we use prefetchers which will use some patterns in history to predict which instructions should be fetched for future execution.

In this paper, we take a look at the different components for cache management, namely replacement policy for the Last Level Cache and the Instruction Prefetcher for the first level instruction cache. We look at the designs proposed in the Instruction Prefetcher Championship as well as the LLC replacement Championship, to gain insight into the different way that these components can be designed. We look at two of them in detail, analyze the designs and try some design space exploration to gain some meaningful insights into the designs. Finally, we take a hand at trying to create a new replacement policy based on a dynamic selection.

The report is arranged as follows:

- Section 2 discusses background work for the areas studied in this paper.
- Section 3 discusses the motivation of the paper.
- Section 4 looks at the details of the Replacement policies and prefetchers.
- Section 5 talks about the evaluation methodology used in this paper.
- Section 6 shows the design space exploration done for the replacement policies analyzed in this paper.
- Section 7 proposes a dynamic design based on set dueling to create a new policy which will perform better than the individual policies.
- Section 8 takes a look at the design space exploration done with this new dynamic design.
- Section 9 looks at the results from this design space exploration and analyzes them.
- Section 10 does a performance analysis of the replacement policies with the prefetchers.
- Section 11 provides the conclusion to this study.
- Section 12 provides the implementation details for this project.

2 Background Work

In this section we look at some of the work previously done with cache replacement policies, instruction prefetchers and a dynamic replacement policy.

In particular we look at two of the instruction prefetchers from the 1st Instruction Prefetcher Championship and two of the replacement policies from the 2nd Cache Replacement Championship.

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.

FNLMMA stands for Footprint Next Line Prefetcher + Multiple Miss Ahead Prefetcher. This design looks at existing literature and the drawbacks of each design and tries to counter those drawbacks by combining 2 prefetchers. The author first proposes the use of a Shadow cache which will be smaller than the Icache, and misses in this shadow cache will trigger a prefetch. Footprint Next Line prefetcher(FNL) prefetches up to K next lines if they are likely to be used in next N instructions. This reduces the amount of prefetches done and does not pollute the cache with unnecessary blocks which will go unused. FNL cannot prefetch blocks which are not contiguous. To work around this, Multiple Miss Ahead Prefetcher is used. The author first talks about a Next Miss Predictor which is used to predict the next miss in the ICache on getting one miss. The MMA is an extension of this which allows the prefetcher to predict not just the next block which will be missing, but the *nth* block which will be missing. This allows timely prefetches and can be used to cover access latency of the second level of cache.

BARCA tries to solve this problem with a different approach. The solution is based on searching a control flow graph. Each node in the graph is a region which contains cache blocks. On certain demand accesses, this graph is searched depth first and the results of this graph are used as candidates for prefetching. The graph is agnostic to branch instructions and treats branching as adding another node in the graph. There is also a shadow cache which mirrors the L1I cache and is used for filtering prefetches as well as deciding the usefulness of the prefetches. They apply a lot of optimizations and tune these parameters to get an optimal solution which performs very well.

The idea of a dynamic replacement policy comes from the paper "Adaptive Insertion Policies for High Performance Caching" by Qureshi et al[5]. The paper proposes a dynamic cache replacement and insertion policy for high performance. It basically exploits the insight that LRU is an effective policy, but it may not work for all workloads, which are larger than the size of the set. In such a case, the instruction may be evicted before it can be reused and will have to go through the full process again, resulting in reduction in performance. The author proposes another insertion policy which places the newly inserted data at the least recently used position- with an understanding that this data will be used only once and therefore can be evicted after it is used. This static method is one way to adapt the cache for a program.

This second policy works well for some workloads where LRU struggles, but then it also performs worse where LRU is better. To reduce this statically, they follow LRU and based on a threshold they place the data in the least recently used position infrequently. However, a better way would be to make this dynamic, so that the policy can adapt to the workload. To do this they create a bimodal policy that feeds the inputs from the cache to both policies and the one with lower misses is chosen to apply. This results in a hardware overhead.To reduce this overhead, they propose the idea of "set dueling", where they allocate a small number of sets of the LLC to each of the policy. Each policy essentially samples those sets. They keep a counter to keep track of which policy has encountered lesser number of misses. The remaining sets are called "follower sets", basically meaning they do not have any set policy for them. Whichever policy the counter points to at that time is the one selected for the set at that time. This works very well for the LRU vs LIP(LRU insertion policy, as named in the paper), since the structures for both the policies are the same and there is very little overhead.

3 Motivation

If we take a look at the competitions from the previous section, we find that the budget for the replacement policy is 32KB whereas the budget for the Prefetcher is almost 128KB. Given that this is the case, it begs the question - which is better to invest in - the replacement policy or the prefetcher?

This paper analyzes the trends in the components, trying to answer this question. We take a closer look at the different designs proposed from the background work, the parameters that influence performance and hardware budget and how we can change them to extract the best possible performance from that. We try to understand the metadata available in the different prefetchers and replacement policies, and their interaction with each other. We also ran some design space exploration experiments to glean meaningful insights from these designs. We take a look at some of the insights achieved from this and analyze our results. We then use this information to tweak our dynamic model to gain the maximum performance from it in a limited budget.

We had a choice to decide between a dynamic prefetcher or a dynamic replacement policy. I discarded the idea of a dynamic prefetcher for the following reasons. The prefetcher uses 128 KB of budget, which is a large budget and performs a lot of intensive calculation like graphs, timestamps and so on. Using two such prefetchers and reducing them to work together within 128KB would surely devolve the performance of both the prefetchers. In addition to that, prefetchers do not have the idea of sampling, as prefetches are done based on the addresses of instructions and used to keep track for future predictions. In contrast, it is better to use replacement policies as the sampling plays a key part in their general mechanisms. Also the sizes are not huge and can be reduced. Typically all

replacement policies will use similar data structures for their working, which allows us to share such structures as well.

Finally coming to our choice of the replacement policies. One issue with the two selected policies is that they largely perform in a similar manner over the majority of the traces. This is a key factor when analyzing our dynamic model, as the dynamic model can only perform as good as the better of the two policies. In case where the one policy beats the other on a trace, I expect the performance of the dynamic policy to follow that policy. However, we acknowledge that the choice of replacement policies might have been better if they had performed orthogonally, which might have led to a better improvement over the two policies.

4 Details of Replacement policies and Prefetchers

4.1 Prefetchers

4.1.1 FNL+MMA

FNLMMA is designed as a combination of two different each of which resolve a different problem. Together they are combined to give a better result. The major components used in this design are an I-shadow cache which is a smaller version of the instruction cache and is used to trigger prefetches.

For the FNL prefetcher, we have two 64K entry tables - Touched and WorthPF. These are 1 bit and 2 bit entries respectively. They are used in the following manner - When there is a I-shadow miss on block B, Touched[B] is set and if Touched[B-1] is set, WorthPF[B-1] also gets set to 3. This gets progressively decremented after a period of L cache misses, and the Touched table entry is reset. This acts as a partial resetting. The prediction algorithm basically says that if on Ishadow cache miss on Block B, WorthPF[B] is not null, prefetch block B+1, and check WorthPF[B+1]. If WorthPF[B+1] is not null, prefetch B+2 and so on. The maximum number of blocks to prefetch used in the design is 5 with a reset period of 8192.

The MMA algorithm uses a 8k entry miss ahead prediction table. Each entry in this table is 71 bits, 12 bits for partial tag, 58 bits for block address and 1 bit for managing confidence and replacement. Apart from this both prefetchers have filter, 16 entry filter with 58 bit entries for MMA to make sure that prefetch candidate chosen has not been prefetched recently, and a 128 entry filter with 17 bit entries for FNL which will work in a similar fashion. In the submitted design, the author uses upto 5 ahead prefetches from FNL and upto 9 miss ahead prediction in the MMA.

4.1.2 Barca

The Barca prefetcher uses a control flow graph to link source and target regions to store information of cache blocks to prefetch on demand accesses. The regions contain blocks of cache which are prefetched. The graph adds a new edge pointing to a node when a new region is discovered. the edges are weighted with a 18 bit counter to count the number of times the edge has been traversed. The graph itself is modelled as a 256 * 64 set associative memory of edges. The tags used are the addresses of the source region. The replacement policy used in this graph is a least frequently used policy.

A search is initiated on a demand fetch, an instruction cache fill or a return instruction. A queue keeps track of recently accessed regions. If a new region which is not a part of this queue is accessed, a depth first search is initiated. The algorithm keeps a running product of the probabilities indicated by the edge weights to make sure that it crosses a certain threshold, else the algorithm considers this to be a wasteful search query. A shadow cache is kept which mirrors tags in the I-cache. This shadow cache is used for filtering useful and useless candidates. For every region searched, all blocks in the region that are not found in the shadow cache are considered candidates.

This algorithm uses a variety of structures to store metadata related to prefetching. They divide regions into a number of blocks, which are varied based on the size of the control flow graph.The ishadow cache stores the tags from i-cache, the tag for each block, a valid bit, a replacement bit for LRU and the address of source block from which it was prefetched. It has its own prefetch queue, from which a fixed number of candidates, in this case 4, are dequeued to the simulators queue at every cycle.

Along with this, the algorithm also uses some optimization techniques. The region addresses are compressed and used to index a 128 entry area table with random replacement policy. As mentioned before return calls are treated as a special case of branches with a particular target, which improves the accuracy of the prefetcher. The authors found that the prefetch queue within the prefetcher often is empty. To reuse this bandwidth, they also maintain a "would-be-nice" queue with lower probability candidates which are dequeued if the prefetch queue is empty, ensuring that the simulator prefetcher is getting the required number of entries every cycle. There is a recently searched list kept to filter out searches for blocks which have been searched in recent memory. Finally, they also tweak the probability counts to make it reflect the usefulness of an edge, not just the true probability. For this, they increment the edge count by 3 when a edge is traversed and is useful, decrement by 2 when it is useless, and increment by 5 when it is useful but late, to reflect that this edge is not only useful, but should be the top candidate so as to achieve timely prefetching.

4.2 Replacement Policies

4.2.1 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. It uses a Signature Hit Count Table, a is_prefetch and RRPV table per line and 64 sampled sets to keep track of history.

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.

4.2.2 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.

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).

5 Evaluation Methodology

For evaluating both the components, we used the simulation infrastructure from the competitions. We used the ChampSim simulator made for the CRC2 with different configurations for replacement with prefetching as well as multicore. We used the ChampSim simulator modified for the Instruction Prefetcher Championship for evaluating the prefetchers.

The prefetcher competition had a fixed set of 50 traces which were used for evaluation of the designs with 50 million instructions for warmup and 50 million for simulation. The cache replacement championship, on the other hand did not have any restrictions on the number of instructions to be simulated, or the traces to be used. In this case, we used a set of 51 traces mentioned in the SHIP++ paper, with the configuration of 10 million instructions for warmup and 100 million instructions for simulation.

6 Results of Replacement policy 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 28 to 40. The graphs show the results here.

7 Design of a dynamic policy

The choice of the policies used in this study could have been better, as both perform in a similar fashion on a majority of the traces. However, we do see some differences in the MPKI for the different traces between the policies as shown in Figure 5. We hope that the dynamic policy proposed in this paper will follow the lower MPKI in all the traces.

To design this, we use the idea of set dueling proposed by Qureshi et al.[5] The general idea is that a few samples of the replacement policy can be used to generalize the behaviour. This idea is also used in both SHIP++ and LIME to generalize the behaviour of a few selected sets. The most straightforward idea for making a dynamic policy is to feed the misses to both policies and choose the better performing policy. However, this is very expensive in terms of hardware. To reduce the hardware budget, the set dueling idea suggests that we should dedicate some sets of the LLC to one replacement policy and some other sets to another replacement policy. We also keep a 10 bit selection counter, which will be incremented every time one of the policy has a miss on its dedicated set, and decremented when the other policy has a miss on its dedicated set. This counter's value will then be used to decide which policy to use for the remaining sets. Since the allocated number of sets will not be large, a big number of the sets will be "follower" sets, decided on the counter.

There were a few caveats when deciding on the design of this new dynamic replacement policy. The first was that the budget was expected to be equivalent to the other replacement policy. To ensure that this is the case, we shrink the budget down to the 32 KB permitted by the competition. In my case, since I was running SHIP++ and LIME together, my original budget was 20KB + 31.2 KB = 51.2 KB with both the policies used as given in implementation. From here, I use some of the insights from the previous experiments to reduce the budget while keeping performance relatively unchanged.

The first thing to notice was that both the policies use a RRPV structure per line to keep track of which line to evict in case of a miss(SRRIP). Since both of them have an overlapping requirement, i decided to share this structure

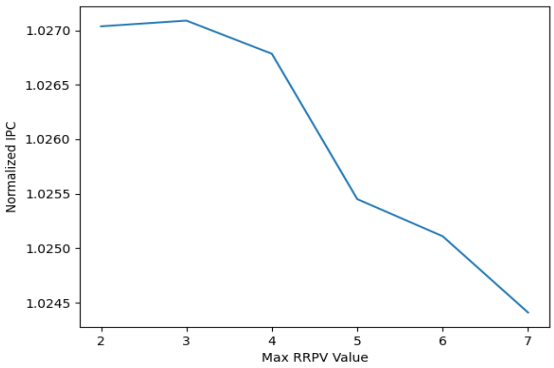

Figure 1: IPC for RRPV high value variations in SHIP++

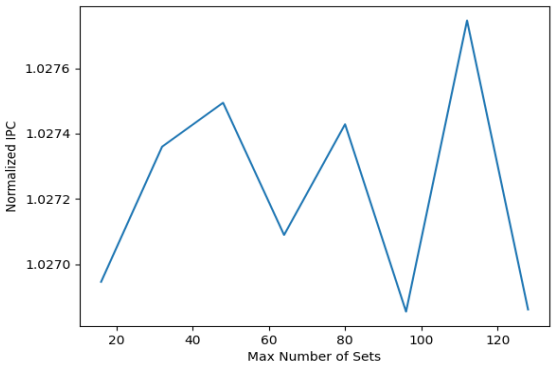

Figure 2: IPC for Variations in the number of sets in SHIP++

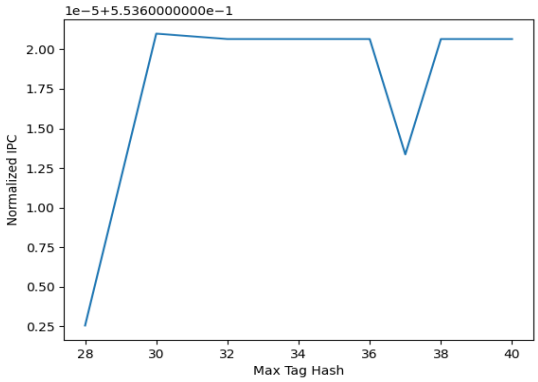

Figure 3: IPC for Variations in the size of the tag in LIME

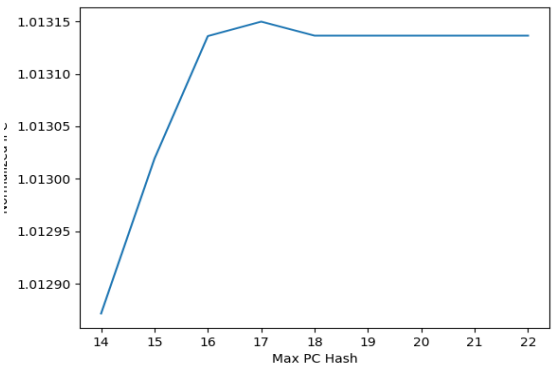

Figure 4: IPC for Variations in the size of the pc in LIME

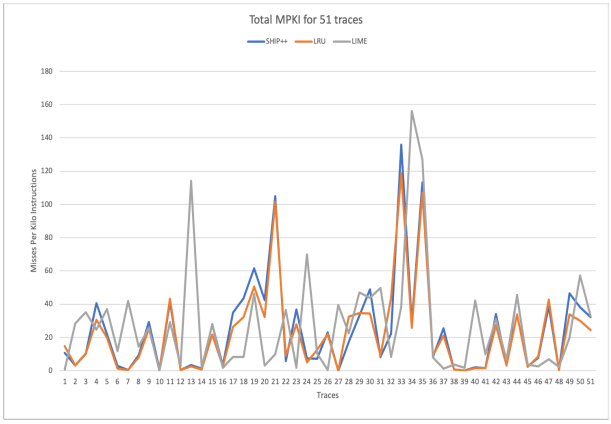

Figure 5: MPKI for SHIP++ and LIME

between the two policies. However, LIME uses a MaxRRPV value of 7 while SHIP++ uses 3. To make it common, I decided to use SHIP++ with a RRPV of 7, as the observation from before is that the IPC is affected in a very slight manner. Doing this effectively reduced my hardware budget from 51.2KB to 43.2KB.

The next step was to see if I can prune the SHIP++ policy a little more. The only other data structure that is taking up significant amount of hardware is the Signature Hit Count Table, which uses hashed 14 bits of PC. However, earlier experiments showed that 13 bits also work just fine with very little drop in performance, but reducing the hardware budget by 3KB. Using this, I now brought the budget down to 40.2KB.

To reduce further, I started looking at LIME. The contributing factors in LIME are the RRPV table, 3 bins used for storing KEEP, BYPASS and RANDOM PCs for future predictions and 20 Belady trainers, each of which use a hashed PC, a hashed Tag, an occupancy vector and a hit vector. Now the occupancy vector is 4 bits long and is dependent on the number of ways, so it cannot be changed and the hit vector is just 1 bit. So I decided to modify the hashed PC and the hashed Tag. From previous experiments I knew that there is no loss in performance up to 30 bits of hashed Tag and the performance is slightly better at 16 bits of hashed PC. So i changed the hash functions to now use these smaller sizes. Since these Belady trainers use a history of 128 and there are 20 such trainers, this reduced the budget a little.

From the scalability experiments done with LIME, we know that the number of trainers at a point does not improve performance much. Also the amount of history from the experiment was optimal at 128 from the three choices in the experiment. To reduce the budget, I decided to use only 100 entries in the history and 15 trainers instead of 20, which brought me to a final budget of 31.485 KB. I also tried out an idea of keeping the history constant and reducing the number of trainers to 10, but that did not provide very good results.

One key point here was that, as mentioned before, both SHIP++ and LIME use some samples of the overall cache to learn behaviour patterns and apply a replacement policy based on that. Since this was the case, I ensured that the dueling sets always include the sets sampled by the policies during initialization. There is a reasoning behind this. Both SHIP++ and LIME rely on their sampled sets to learn the patterns in the workload. However, if these sampled sets are not dedicated to running that particular policy, there is a possibility that a set which is used for sampling by SHIP++ is dedicated by chance to LIME, or the selector says that the current prevalent policy is LIME at the time, which would render that set ineffective and reduce the performance of SHIP++ as a whole. Since LIME used only 15 trainers in my model, I dedicated 49 more random sets to LIME, so that the counter is not biased towards SHIP++ due to the sheer number of sets dueling for SHIP++.

The other design decision I made was that instead of using a general random distribution of sets as used by SHIP++, I ensured random distribution over the 2048 sets. I effectively chose a random number at every 32 sets which would be dedicated to SHIP++ and another which will be dedicated to LIME. The 15 trainers are chosen from these 64 at a similarly fair distribution.

8 Design Space exploration and analysis

The previous section showed the use of previous insights to improve the design and bring the design under the required budget of 32 KB while maintaining performance. Since I had already conducted experiments on the replacement policies individually, I did not conduct any new experiments except for those mentioned before to reduce the size. Instead in this case, I varied the new parameters, the counter and the number of dueling sets.

I varied the counter to 2 different values. I made it 11 bit to count up to 2048 and reduced it to 9 bit to count up to 512. With the reduced count, the performance reduces, while the increased count keeps is almost at an equal. It seems that for 64 sets, which was my baseline model, the best counter is 1024, which was also the design choice made in the set dueling paper by Qureshi et al[5].

The other variation I made was changing the number of sets which are dueling from 32 to 128. While choosing 32 sets, 32 sets of SHIP++ were randomly assigned to follower sets, which would result in an effective loss in performance. I also increased the dueling sets to 96 and 128, to see the effects. In this scenario, the SHIP++ sampled sets selected as every 2 of the 3 and every second set from the dueling sets. The graphs shows the performance of these experiments.

9 Results and analysis

From figure 6, we see that as we vary the number of sets, 64 is the optimal number of sets to be used for the dynamic sampling. It does perform a bit worse in the 32 sets, but that is likely due to the fact that SHIP++ has only half of its samples available, which reduces the generality of the overall algorithm. As we increase the number of sets, the performance drops further. This is because now there are many extra sets which are contributing, but they themselves are not being used to sample and generalise. As a result these are just sets that have no useful information coming in except that they belong to a particular policy, so it is added noise.

The selection counter is just used to signify the confidence in a particular policy based on the number of misses on the policy. I varied this counter to be 3 different values - 512,

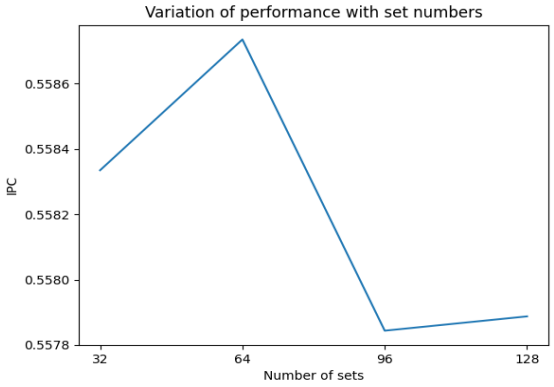

Figure 6: IPC for variations in number of sets

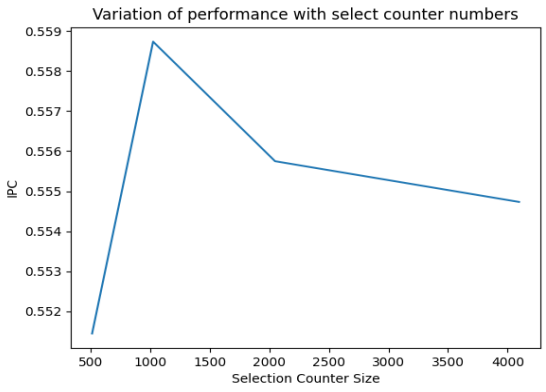

Figure 7: IPC for selection Counter

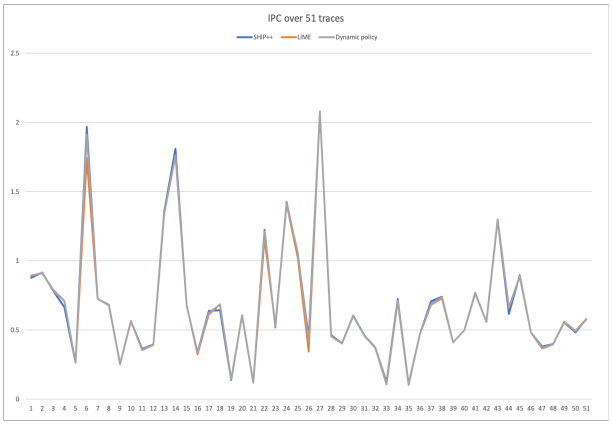

Figure 8: IPC for all traces

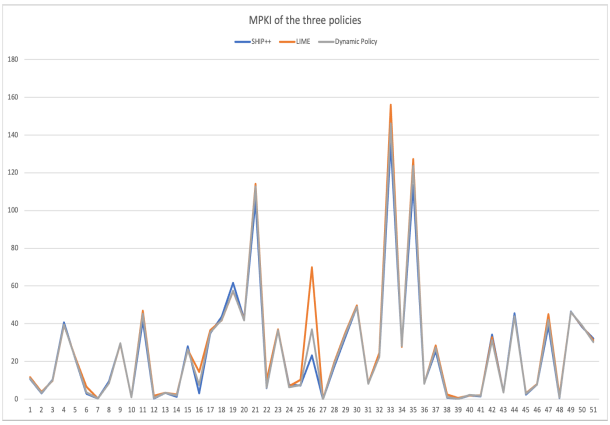

Figure 9: MPKI for All traces

1024, 2048. The results show that 1024 is a good size for the counter. It allows a reasonable amount of confidence for either policy without having to go through an extremely large number of instructions to make a switch to another policy. Having 512 seems too low, since the counter can easily switch from one end to another.

The overall IPC and MPKI of the dynamic policy in comparison with the two policies it combines is shown in figures 8 and 9. These graphs shows the performance in terms of IPC and MPKI for all the policies on a per trace basis.

From this figure we can see that the dynamic policy closely follows the better performing of the two policies. However, in some workloads, due to the second policy dueling, there seems to be a slight bit of noise, which makes it so that it does not equal it exactly. We can see for trace 18 LIME is better than SHIP++ and the dynamic policy follows LIME instead of SHIP++. Table 1 shows the normalized IPC improvement for the three policies over a baseline of LRU. We see that our dynamic policy is better than LIME by around 1.1%, but is slightly lesser than SHIP++. SHIP++ provides a speedup of around 2.7% while our dynamic policy gives a speedup of 2.25%.

As discussed above, the reason for this is that LIME introduces some noise into the system which slightly reduces the performance. And as SHIP++ is the better performing

policy in most of the cases, SHIP++ comes out to be better. It is possible to perhaps beat both policies if the two policies performed very orthogonally for different traces, i.e. Policy 1 performs very well while policy 2 performs very poorly and vice versa. In such a case set dueling could offer a better result. We also see them in the graph. We once again see that our dynamic policy follows the lower MPKI. The general idea behind set dueling is sound and working fine. The only issue is the fact that the choice of replacement policies could have been better to provide a better performance boost.

10 Performance analysis and comparison with prefetcher

Now that we have talked about a new dynamic policy for cache replacement, let us cast our thoughts back to the the motivation behind this paper. The motivation was to find, which gives a better performance, a prefetcher or a replacement policy. To that end, we look at the results obtained from the prefetchers here and then compare them along with their budgets to the prefetchers.

Here are the results from the baseline implementations of the Instruction Prefetcher Competition for the two designs I analyzed - Barca and FNL+MMA. These two prefetchers take different approaches on prefetching. Barca uses a control flow graph to keep track of all the possible outcomes from a particular address which is modeled as a node. It prefetches these results and keeps them in the prefetch queue, from where they are brought into the instruction cache.

The other prefetcher I analyzed, FNL+MMA combines 2 prefetchers into a single model, both of which do a relatively simple single function. The idea behind this was to look at prefetching as a set of 2 problems and use these two solutions to solve these two problems. It uses the FNL prefetcher which prefetches the next 5 address after the current address. The MMA prefetches addresses which are more distant at a distance of 9 or beyond, which might be useful for the current address. Together these prefetchers work in tandem to provide a pretty good performance.

The graph in figure 10 shows the IPC speedup of the two prefetchers over a baseline of no prefetcher. This shows us that prefetchers have a massive gain in performance. However, a direct comparison with a replacement policy is not apt, since the budget allocated to these policies is also much larger compared to the replacement policies. To compare this, we take the best performing replacement policy, which is SHIP++, compare the performance speedup/KB of budget with the better performing prefetcher, which is FNL+MMA, and compare the performance speedup per KB of budget.

Here we see that the prefetcher provide a performance gain of almost 29% over a processor with no prefetcher. To gain an understanding of the IPC speedup per KB of budget, we divide IPC speedup of the best prefetcher, which is FNL+MMA, by its reported budget, which is approximately 95KB. We then take a look at the replacement policy. The SHIP++ provides a gain of 2.7% while our dynamic policy provides a gain of 2.25%. The SHIP++ is the best replacement policy here. SHIP++ has a budget of 20KB for the examined configuration. Based on this, we see that the speedup per KB of FNL+MMA is more than SHIP++. FNL+MMA provides nearly 0.31% speedup per KB as compared to 0.135% speedup per KB of SHIP++. Based on this result, a prefetcher is definitely the better place to invest money in to gain better value for money.

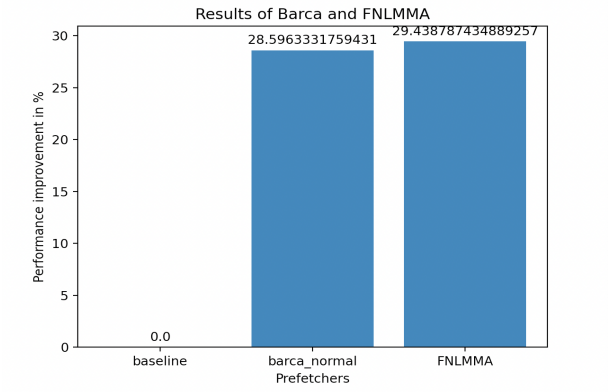

Figure 10: IPC Speedup for Prefetchers

11 Conclusion

In this paper we took a look at different techniques to improve cache performance. We explored cache replacement policies and instruction prefetchers which improve the data and instruction caches respectively. We also took at adaptive cache replacement policies, and tried to create our own policy based on the idea of set dueling. Our idea performed as well as it could be expected to perform based on the design choices. We used the insights gleaned from Design Space Exploration of the Cache replacement policies to improve the performance of our dynamic policy while keeping it under budget.

Finally, we addressed the question of whether it is better to invest in a prefetcher or in improving a replacement policy. We find that FNL+MMA provides a better speedup of 0.3% per KB. However, it is worth noting that the speedup of FNL+MMA is computed over a baseline of no prefetcher, while the SHIP++ improvement is computed over a LRU baseline. The prefetcher may provide a lesser speedup with a prefetcher already existing. However, instruction prefetching is a relatively new problem still in its exploratory stage, so I believe that investing in a better prefetcher should result in a better performance upgrade as compared to a replacement policy.

12 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 Sim3. 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.

13 Acknowledgments

I would like to thank Dr. Samira Mirbagher Ajorpaz for providing this wonderful opportunity to create a custom design based on existing designs and guiding us in this assignment. The whole study is done using the infrastructure provided by the 1st Instruction Prefetching Championship and the 2nd Cache Replacement Policy. I use their simulator for simulating all the prefetchers and replacement policies, 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.

14 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.

[5] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07). Association for Computing Machinery, New York, NY, USA, 381–391.