HW1 Simulation Assignment Report for CSE240C UCSD Winter 2022

Monil Shah A59012111

ABSTRACT

With increasing memory wall, it becomes crucial to improve the Cache Hits. Memory Latency can be hidden through multiple means. Out-of-Order execution is one such paradigm however it would still reach a bottleneck in fetching new set Instructions. Prefetchers alleviate this problem by bringing the likely to be used Instruction/Data Blocks before it is actually used. This paper presents design analysis of two of the 3 Winner Prefetchers at the 1st Instruction Prefetching Championship based on the Simulator ChampSim.

1 Introduction

With growing instruction-working set among neural network, cloud and server applications, instruction foot print is ever increasing and so is instruction cache misses. Instruction prefetchers effectively hide cache miss latency by prefetching the blocks speculatively. This is done in separate buffers so as to not affect the normal cache operation through cache pollution or conflict misses. If the data isn't used there is no additional penalty in comparison to not using prefetchers. This paper is organized as follows : Section 2 provides high level summary of 3 winner prefetchers at the 1st Instruction Prefetcher Championship, Section 3 lists important aspects of two of the Winner Prefetchers, Section 4 Provides simulation results for Baseline Prefetchers and some design space exploration of its parameters. Section 5 identifies some first order structures and analysis over different hardware budgets, Section 6 is conclusion followed by Section 7 which contains the Code and Scripts involved in running these simulations

2 Literature

2.1 D_JOLT

D-JOLT predictor [1] is a return address stack based prefetcher. The return address stack is used to predict the return address of the next set of instructions. During the learning phase it generates a signature based on the Hash of Return Addresses in the Stack and associate with corresponding cache miss that was observed. It is based on a RDIP prefetcher with the only difference being, the Stack of Return Addresses uses all addresses instead of evicting them along with a counter to distinguish the signature, this helps in adding some correlation between previous function calls. D-Jolt uses a total of 3 prefetchers, each with its own unique property to compensate for the accuracy/ timeliness drops of other prefetchers. The long and short prefetchers use tables to record signatures and miss addresses. Fallback prefetcher is a stream based prefetcher [5].

2.2 FNL+MMA Prefetcher

FNL-MMA Prefetcher [2] uses a combination of two prefetchers. The FNL prefetcher is based on the idea that all next lines should not be prefetched, as it causes a lot of L2 accesses and over-pollution. Instead only meaningful and reasonable to be used lines should be prefetched. By associating the current miss with previous misses and a 2bit counter, the next N blocks are fetched. The corresponding next miss need not prefetch again and uses a filter mechanism to reduce prefetches. The MMA prefetcher prefetches non-contiguous lines and tries to foresee several blocks ahead in what instruction blocks are likely to be missed based on the current misses. By associating confidence with effect of one miss on another, it can look ahead into misses.

2.3 Temporal Ancestary Prefetcher

This prefetcher is developed on the concept that a lot of cache blocks are not frequently used and are termed as dead if it will not be referenced again before eviction. The TAP prefetcher uses this to the advantage to prefetch the next set of blocks that can be replaced before the data is actually evicted and replaced. It consists of an ancestary table that tracks what addresses were seen on accessing current PC. Depending on which address is accessed a counter bit is incremented. The prefetcher works alongside next-line so as to reap benefit of next prefetching as well as faster replacement instead of relying on demand accesses. On a cache access the ancestary table entry is read and its prefetches are requested depending on counter. To compress the data few significant bits of Tags are stored in a shadow cache for faster and parallel access

3 Meta Data

3.1 D_JOLT

3.1.1 Meta Data - The D_JOLT prefetcher uses 3 prefetchers. The long range prefetcher has a signature queue consisting 15 deep 23-bit wide entries. The signature Generator consists of a 7deep 32-bit wide FiFo and a 32-bit Counter. The miss table consists of 2048 set, 4way associative deep 76-bit(Tag, LRU, miss vector) wide entries. The short range prefetcher has a signature queue consisting 4 deep 23-bit wide entries. The signature Generator consists of a 4 deep 32-bit wide FiFo and a 32-bit Counter. The miss table consists of 1024 set, 4way associative deep 77-bit (Tag, LRU, miss vector) wide entries. The fallback prefetcher uses 16 deep 65-bit wide (Tag, replacement, address) entries Train table and 16 deep 63-bit (Tag, replacement, address) wide entry Monitor Table. There is a upper bit table consisting of 31 entries and 41 bit wide (Tag, Valid). There is a miss table of 256 set, 4 way 79 -bit wide (Tag, LRU, miss vector). All totalling to 125KBytes

3.1.2 Key Design - The key design of the prefetcher is the signature generator, the miss tables and the signature queue. The fifo stores return addresses from the stack and implements a counter that counts number of returns to associate a unique signature instead of limited signature.. Long range prefetcher has 7 deep FIFO and short range has 4 deep. The signature queue implements the distance parameter. By adding the signatures in a queue of N-Deep, the prefetcher associates a miss address on popping a signature from queue, which would be N-deep in the past. Miss table associates signature with Miss address. Upper bit table is added as a fully associative cache to reduce size of normal miss table.

3.2 FNL+MMA

3.2.1 Meta Data There are several key components. I-shadow cache with 192 entry 17-bit wide Table that is similar to a Icache used to trigger prefetch. It uses a 64K entry 1-bit Touched and 2-bit WorthPF Tables. These are used to predict which blocks to fetch. FNL prefetcher uses a 128 entry 17 bit FNL filter to skip requesting already present blocks in prefetch buffer. MMA prefetcher uses an 8K entry Miss table which is 71 bit wide (tag, block-address, control bits) and a 24 entry 58bit wide MMA filter. Total hardware budget lies around 96KBytes. And FNL filter reset interval of 8K. The FNL Prefetcher uses a distance of 5 blocks and MMA uses a distance of 9 blocks. There is a slight difference in the paper submitted to the conference vs the actual code used. The IPC speedup achieved with Old parameters (9 distance and 16 entry MMA filter) is 1.28745 whereas with the updated code (11 MMA distance and 24 filter) is 1.292. We will refer the FNL+MMA in this paper as the Prefetcher with the updated parameters

3.2.2 Key Design - The touched Entry table is a 1bit counter of recently touched block flag along with a counter to indicate its demand access in a dedicated fix interval. The Touched Entry is set when a miss occurs, and the corresponding counter of previous Block is set to 3, to indicate that a miss on B causes miss on B' . By changing the confidence counter of miss intervals, it correlates when to prefetch and when not to. Next time when a Block Miss occurs, depending on the Counter, next 5 blocks are prefetched. The Miss interval dictates the time to refresh flags and clear Recentness, to only fetch meaningful data. The MMA Prefetch Table associates the miss address with predicted address using a Cache like table of Tags and Block Address. The MMA prefetcher prefetches 9 blocks in advance since we need to prefetch blocks well in advance. This is slightly different from a Next predicted Miss prefetcher as it can look N blocks ahead of potential miss.

4 Simulation Methodology

The simulations are run using the Prefetching Championship Infrastructure present at [4]. Traces are the same as run for the championship. While some traces are hidden, in this paper we only check with the traces that are publicly available. There are total 50 available traces which are a mix of 35 Server, 8 Client and 7 SPEC Benchmarks. The prefetchers are warmed up for 50M instructions and then run for 50M instructions. Results are then populated as seen in the graphs. Initially for Baseline Model without the Instruction prefetcher all 50 traces are run and simulated. Same is done with Prefetchers D_JOLT and FNL with their Baseline to reproduce the result.

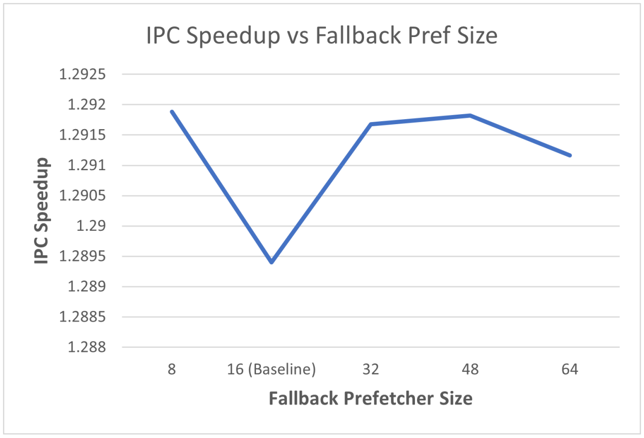

Figure 1: IPC Speedup with varying entries of DJOLT Fallback Prefetcher

The IPC Speedup attained from D_JOLT Prefetcher over Baseline is 1.2894, and MPKI reduction is 93.62%. The IPC Speedup attained from FNL+MMA Prefetcher is 1.2924 as apposed to 1.287 in paper (owing to updated code) and MPKI to be 90.93% as opposed to 91.8% in the paper. Comparing to the original Result in Paper, the error margin in IPC is 0.42%. Following sections describe the Design Space Exploration of the Prefetcher Structures and are simulated with all 50 traces to get accurate results.

5 Design Space Exploration

5.1 D_JOLT

The Parameter chosen for Design space exploration are Fallback Prefetcher Entries, Distance of Long Prefetcher and Distance of Short Prefetcher. Fallback prefetcher is important and gives confidence into either of the prefetchers as well as prefetch when none of the other prefetchers are fully trained and hence is one of the good parameter to check. Short and Long prefetchers are wide enough already and have associativity enough to handle conflict/capacity misses. So their efficiency is more determined by the distance they look ahead to fetch the blocks.

5.1.1 Varying Entries of DJOLT Fallback Prefetcher - In this exploration, we vary the Size of the Fallback prefetcher Training and Monitor Tables and see that Baseline of 16 entries gives the minimum IPC speedup as seen in Figure 1. The ideal trend is to have optimal performance at 16 entries and decreases on either sides of the Baseline. Shorter Entries lead to more conflicts and more entries are not that meaningful since the Prefetchers are warmed up with enough instructions. The trend of decrease from 8 to 16 can be attributed to the fact that x264 and perlbench Workload suffers more than other workloads from higher than 8 entries, this is due to the fact that both short and long prefetcher are supplemented with inverse support from Fallback prefetcher due to higher entries being present and less optimal entries that can give closer prediction.

5.1.2 Varying Distance of DJOLT Long Prefetcher - The trend expected in this exploration is to have an opti-

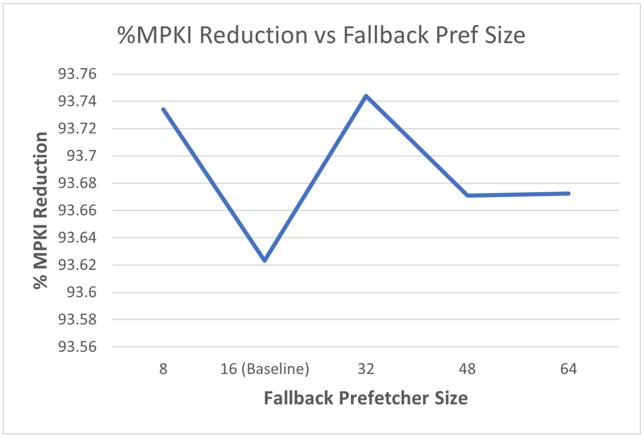

Figure 2: % MPKI reduction with varying entries for DJOLT Fallback Prefetcher

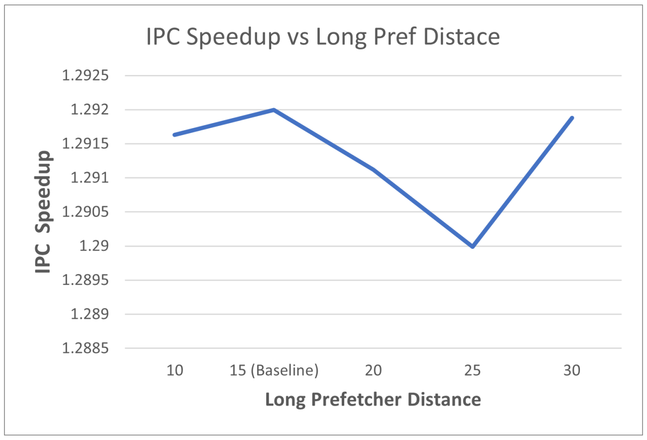

Figure 3: IPC Speedup with changing Distance for DJOLT Long Prefetcher

mal result at Baseline and lesser MPKI reduction (Fig 4)/ IPC improvement (3) as we increase the distance as farther fetches may be more probable to be wrong. However the IPC dip is huge in case of 25 distance ahead. This is seen in server workloads as increasing distance results in sub-optimal prefetch, whereas some smaller applications like Go are achieving higher improvement at very large distances due to converging of execution path for bigger procedural blocks. MPKI reduction trend looks as expected. WIth higher Distance, the prefetches are redundant and cause more conflicts in the cache with contention to L2 accesses.

5.1.3 Varying Distance of DJOLT Short Prefetcher - The trend expected in this exploration is to have an optimal result at Baseline and lesser MPKI reduction (Fig 6)/ IPC improvement (5) as we increase the distance as farther fetches conflicts with Fetches of Long distance prefetcher and since we have lesser prefetches of immediate blocks. Very close blocks will take time before they can actually be utilised and hence 4 as baseline distance seems a feasible breakpoint

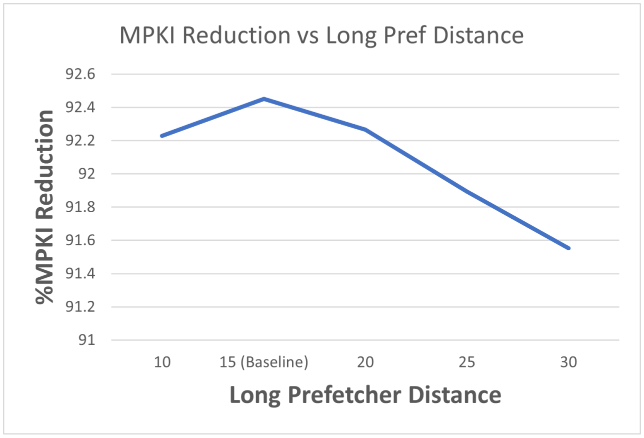

Figure 4: %MPKI reduction with varying distance for DJOLT Long prefetcher

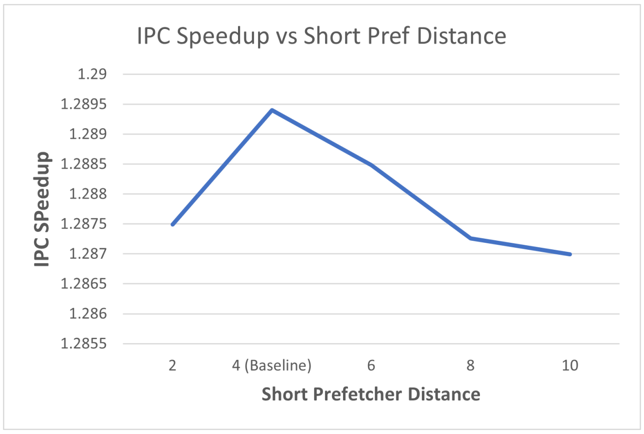

Figure 5: IPC Speedup with changing Distance for DJOLT Short prefetcher

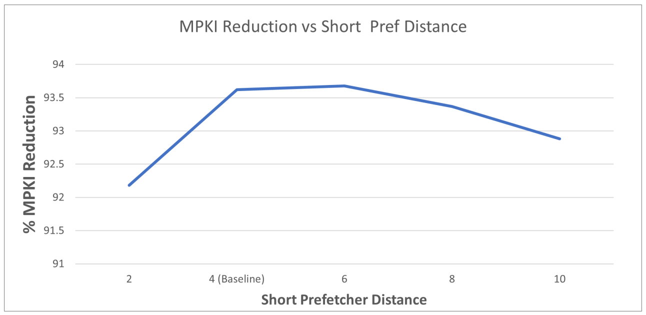

Figure 6: %MPKI reduction with varying Distance for DJOLT short Prefetcher

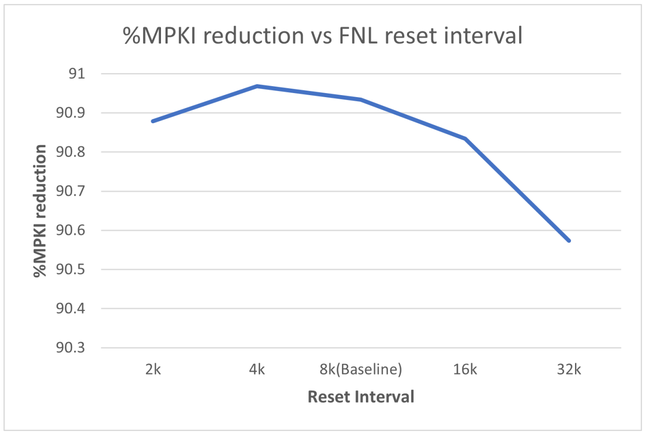

Figure 7: MPKI reduction with changing Reset Counter for FNL Prefetcher

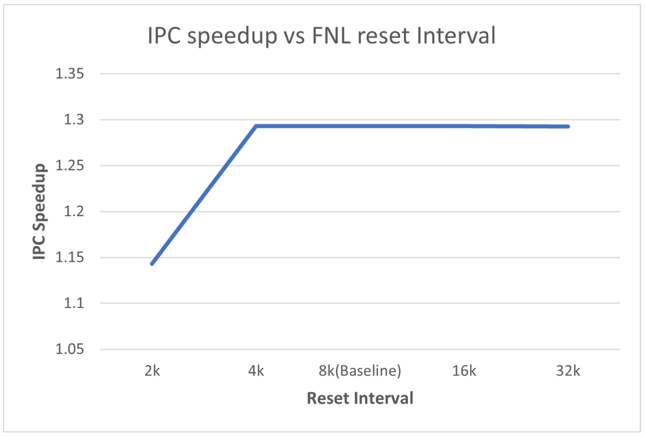

Figure 8: IPC Speedup with changing Reset Counter for FNL Prefetcher

5.2 FNL+MMA

Of the Parameters within the FNL and MMA filter, the ones that seemed more important were the Reset Interval, MMA Distance and the MMA Filter. The reset interval of FNL determines how crucial is the information and how relevant is the confidence information over certain period of time. MMA prefetcher is what gives FNL prefetcher an additional boost in prefetching by guessing farther misses, hence the blocks can prefetch without being replaced before usage is a good metric of its performance. Similarly if MMA keeps fetching blocks, it can saturate L2 bandwidth and leave no room for demand access hence the number of filter entries can help to reduce L2 accesses.

5.2.1 Varying Reset interval of FNL Prefetcher - The expected trend is that with Higher Reset intervals there should increase and thereafter show minimal or low improvements as the confidence in correlation may be stale. As it is expected, the MPKI reduction (Figure 8) reduces as the reset interval is increased. Similar is with IPC improvement (Figure 7) and it becomes saturated with higher interval as the prefetches being requested aren't meaningful. There is practically no returns. However the interval used in the paper is 8K whereas the sub optimal interval from simulations seem that it should have been 4K. This is confirmed by running through all traces

5.2.2 Varying Distance of MMA Prefetcher - MMA is creating requests of farther misses in advance, but the expectation is to have diminishing or inverse returns with farther block prefetch as it is likely to be replaced by blocks from FNL or they are simply not useful. This is similar to the MPKI reduction in Figure 10 and IPC improvement in Figure 9 . However there is a sharp decrease in distance of 15 blocks ahead, this might be possible due to procedure blocks being typically sized between 10-18 blocks and that lot of requests are falling within the range of Procedure instruction boundaries and encountering Branches/Returns.

5.2.3 Varying Filter Number of MMA Prefetcher - By varying the Filter size but keeping the Distance of Prefetcher constant, the speedup and MPKI should increase till opti-

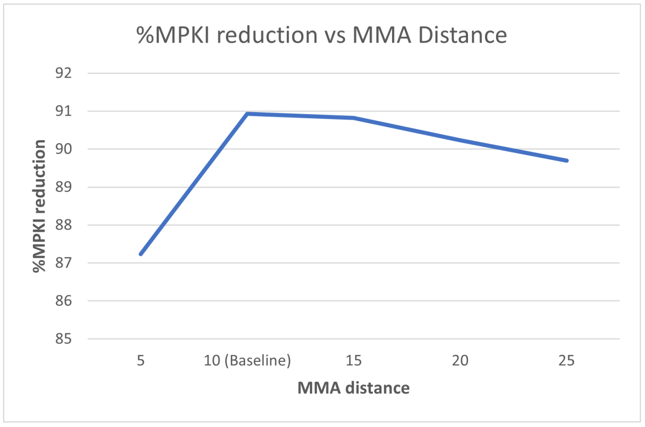

Figure 9: %MPKI reduction with changing MMA Distance for FNL5+MMA Prefetcher

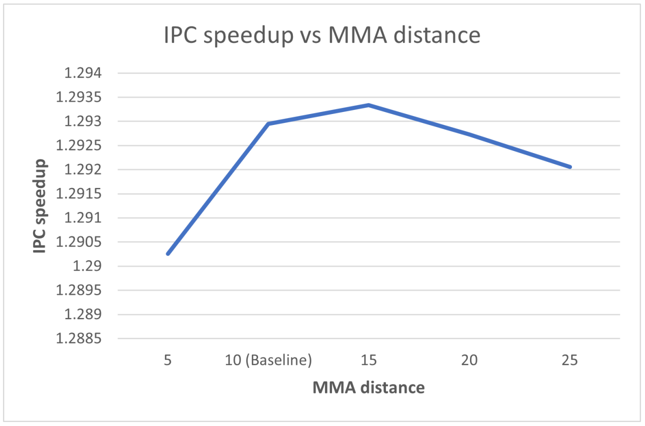

Figure 10: IPC Speedup with changing MMA Filters for FNL5+MMA Prefetcher

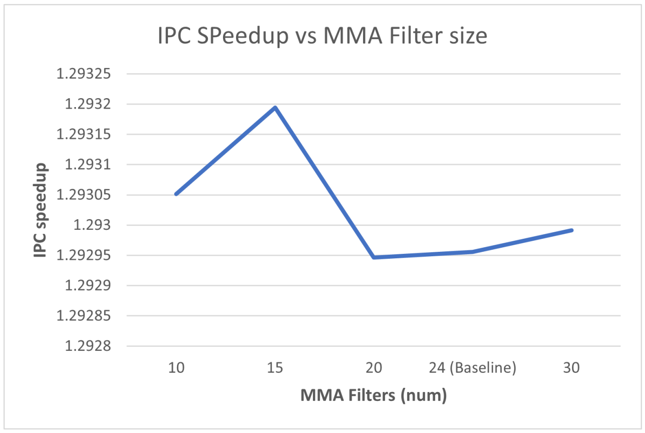

Figure 11: IPC Speedup with varying Distance for MMA Prefetcher in FNL+MMA Prefetcher

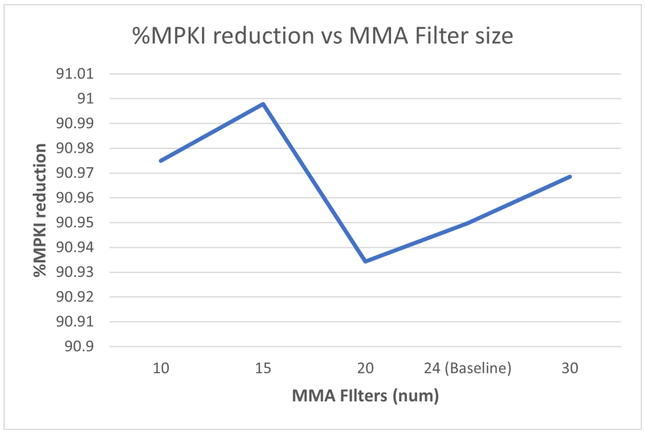

Figure 12: %MPKI reduction with varying Filter number for MMA Prefetcher in FNL+MMA Prefetcher

mum setting and decrease beyond an optimal setting. The IPC speed up figure 11 and MPKI reduction in Figure 12 seems to be otherwise from sharp decline at 20 Filter size and marginally improving with higher distances. This can again be validated by previous statement that procedural blocks are typically 10-18 blocks in length and higher blocks lead to wasteful prefetches around the boundary, more prefetches than the block length of 18 actually benefits as some additional prefetches beyond the mispredicted ones have been requested and will benefit when used. Meaning only certain blocks from the filter would need to be invalidated on misprediction that are near the boundary, anything before and after that seem to be useful.

6 Dealing with Hardware Budget

IPC Championship stipulates a budget of 128KBytes. However here we have simulated the performance of the Prefetchers at smaller and larger hardware budgets and see performance. Ideal expectation is to have increasing performance beyond the current limit upto a optimal value and then saturating or diminishing returns.

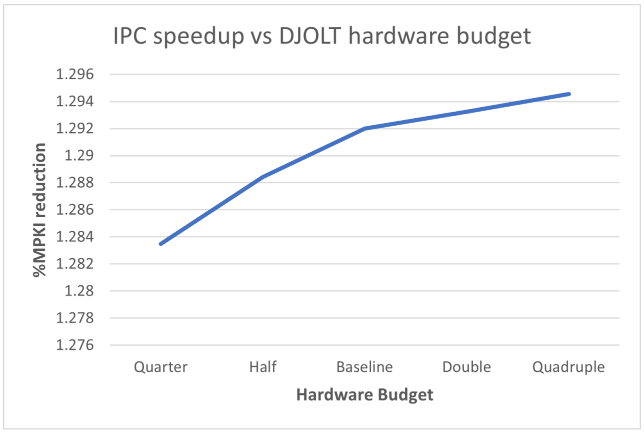

Figure 13: IPC Speedup with changing Hardware Budget for D_JOLT Prefetcher

6.1 D_JOLT

D_JOLT prefetchers has multiple Tables which are the first-order structures in the design, like the Long Prefetcher miss table, Short prefetcher miss table and Extra miss table. The current hardware cost is 125KB, and tables were reduced/increased in multiples of 2 to simulate as shown in Figure 13 and 14. One thing to note for lower size tables is signature bits had to be reduced to fit within the index. Signature bits used for 64kB budget is 22 and for 32kB budget is 21. Only table entries are updated but not the Tag and associativity, so as to not loose information for lookup. The configuration for each sizes are as follows:

- 32 KB Budget- 64 Fallback (2.5kB), 256 Short(10KB), 512 Long Prefetcher Sets(19KB). Actual utilization - 31.5kB
- 64 KB Budget- 128 Fallback(5KB), 512 Short(19KB), 1024 Long Prefetcher Sets(38KB). Actual utilization - 62kB
- 128 KB Budget- 256 Fallback(10KB), 1024 Short(36KB), 2048 Long Prefetcher Sets(76KB). Actual utilization - 122kB
- 256 KB Budget- 512 Fallback(20KB), 2048 Short(72KB), 4096 Long Prefetcher Sets(152KB). Actual utilization - 244kB
- 512 KB Budget- 1024 Fallback(40KB), 4096 Short(154KB), 8192 Long Prefetcher Sets(304KB). Actual utilization - 498KB

6.2 FNL+MMA Prefetcher

FNL prefetchers has 3 Tables which are the first-order structures in the design, the Touched and WorthPF Tables of FNL and miss table of MMA. The current hardware cost is 96KB, and tables were reduced/increased in multiples of 2 to simulate as shown in Figure 15 and 16. Only table entries are updated but not the Tag/ associativity / Counter Width, so as to not loose information for lookup. The configuration for each sizes are as follows:

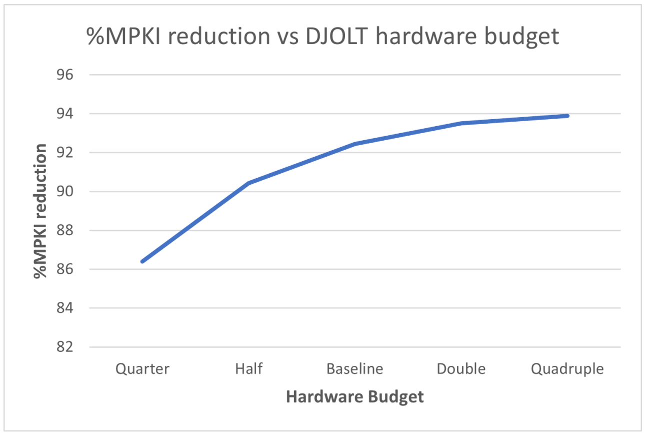

Figure 14: %MPKI reduction with varying hardware budget for D_JOLT Prefetcher

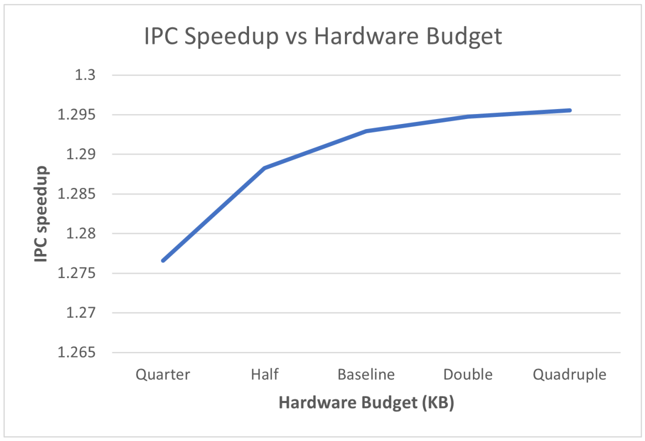

Figure 15: IPC Speedup with changing Hardware Budget for FNL5+MMA9

- 32 KB Budget- 16K Touched and WorthPF Entries (6KB), 2K Miss table (71bits - 18KB) - 24kB Actual
- 64 KB Budget- 32K Touched and WorthPF Entries(12KB), 4K Miss table (71bits - 36KB) - 48kB Actual
- 128 KB Budget- 64K Touched and WorthPF Entries(24KB), 8K Miss table (71bits - 71KB) - 96kB Actual
- 256 KB Budget- 128K Touched and WorthPF Entries(48KB), 16K Miss table (71bits - 142KB) - 190kB Actual
- 512 KB Budget- 256K Touched and WorthPF Entries (96KB), 32K Miss table (71bits - 284KB) - 380kB Actual

6.3 Analysis

6.3.1 Hardware Cost Comparison - It is evident from the simulations that FNL+MMA prefetcher is under utilizing the Hardware Budget while still making significant performance improvement as compared to D_JOLT. The Budget utilization at 512kB by scaling the first order structures is 75% for FNL+MMA and 97% for D_JOLT. While MPKI reduction

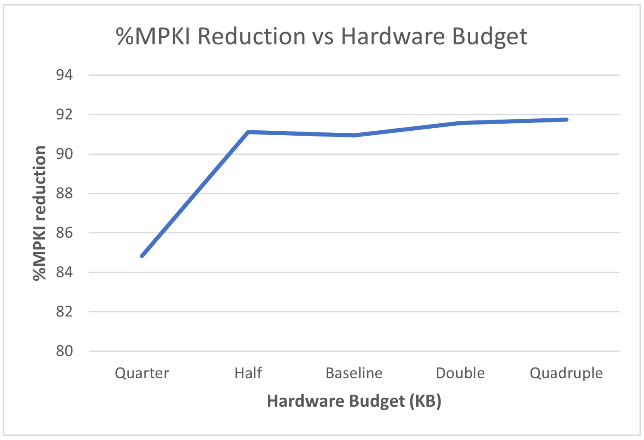

Figure 16: %MPKI reduction with varying hardware budget for FNL5+MMA9 Prefetcher

for FNL+MMA is slightly lesser than D_JOLT, FNL+MMA results in higher accuracy in terms of IPC improvement and shows signs for more improvement at higher budgets. Under more constraints like Area cost, FNL+MMA is likely to perform better. FNL+MMA performs better than D_JOLT at lower budget due to the following : 1. FNL+MMA fetches blocks farther in advance that are likely to be missed later. So it associates single miss with prefetch of 2 blocks of different distances. 2. D_JOLT correlates more information than FNL+MMA but due to the usage of 3 prefetchers, the accuracy suffers where one of the Prefetcher is incorrect but has more priority in deciding the prefetch request. 3. By using a filtering mechanism, FNL+MMA reduces the prefetch blocks that are requested and potential reduction in Prefetch buffer occupancy, thereby helping with reducing conflict.

6.3.2 Timeliness - In terms of Timeliness, D_JOLT might be better than FNL+MMA for 2 reasons. 1. The fully associative search of the Elements in the FNL or MMA Filter can take up multiple cycles, whereas the D_Jolt Prefetcher only does a lookup of upto 4 ways in a set while accessing the cache for block address 2. The MMA and FNL together can look around 11-15 blocks in advance (depending on 5 blocks that it can fetch), but the distance of the D_Jolt long prefetcher itself is 15 and it can look ahead upto maybe 4blocks which makes it timely in terms of supplying the block when it is needed instead of being evicted. 3. The signature generator is not on the critical path as it is stored in FiFO and the Popped element is used to index the Tables. Same with FNL+MMA, the computation of address is not on critical path.

7 Code

The code is available in repository : CSE240C-Winter2022
Steps to run are added in File called StepsToRun.txt

8 References

[1] https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/D-JOLT.pdf

[2] https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/FNLMMA-final.pdf

[3] https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/tap_final.pdf

[4] https://github.com/ChampSim/ChampSim

[5] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers,” in HPCA, 2007, pp. 63–74

HW2 Simulation Assignment Report for CSE240C UCSD Winter 2022

Monil Shah A59012111

ABSTRACT

With increasing memory wall, it becomes crucial to improve the Cache Hits. Memory Latency can be hidden through multiple means. While we cannot have infinite size caches to avoid conflict misses, we can improve upon cache replacemeent policies to make sure conflicts are reduced. This results in better utilization of memory bandwidth by not thrashing the memory controller with accesses.This paper presents design analysis of two of the 3 Winner Prefetchers at the 3rd Cache Replacement Championship based on the Simulator ChampSim.

1 Introduction

With growing instruction-working set among neural network, cloud and server applications, instruction foot print is ever increasing and so are cache misses. To reduce cache thrashing, one of the ideas is to prefetch necessary data, whereas other is to have a better replacement policy. With better replacement policies it is possible to predict which blocks are likely to be reused later and evict entries that are less likely to be reused later. This paper is organized as follows : Section 2 provides high level summary of 3 winner prefetchers at the 3rd Cache Replacement Championship, Section 3 lists important aspects of two of the Winner Prefetchers, Section 4 Provides simulation results for Baseline Policies and some design space exploration of its parameters. Section 5 identifies some first order structures and analysis over different hardware budgets, Section 6 is conclusion followed by Section 7 which contains the Code and Scripts involved in running these simulations

2 Literature

2.1 Expected Hit count Replacement

The paper tries to correlate reuse distance with expected Hit count of a Block. Higher the remaining expected Hit count, lesser the reuse distance. The idea needs a baseline replacement policy to get the number of hits which is DRRIP for them since it has low overhead giving more space to implement their predictor. They remember the hit count of the cache block for past 2 insertions. And the average of it is used as EHC. The paper leverages to combine Tag information on the basis of less unique tags in LLC based on previous study to reduce their history storing overhead. The history information consists of a 128 entry 16way table with 1 valid, 2 3-bit counters, an LRU recency bits and Tag bits. The metadata and counters are updated on saturation of Hit counters or eviction on non-saturated counter.. Depending on if the Tag matches, a new line is issued or previous Count is pushed ahead and next count is added back. The victim is selected depending on the RRPV value for the block in the cache, the EHC information and the current Hit counter of the block. By Subtracting Current Hit counter and RRPV value from EHC, the lowest value block is evicted. This is based on the fact that RRPV highest was anyway supposed to be removed and it making closer to zero value indicates that the expected number of hits of that block is less. [3].

2.2 Hawkeye Replacement Policy

This paper is based on modifying the existing Hawkeye replacement policy. Earlier policy treats demand and prefetch requests in same way, but the performance impact of both will be different depending on the workload, so to optimally decide on the replacement, the paper uses separate tables for demand and prefetch requests. The paper tries to construct Belady's algorithm by associating Load instruction with being cache friendly or cache adverse. Friendly lines are given higher priority to inert into the cache. By associating the liveness of blocks in cache in the past with future access, it determines what should be the occupancy of the blocks in the future. The Hawkeye predictor identifies the lines to be friendly or adverse depending on confidence counter. It works in correlation with Optgen to strengthen the confidence in previous PC if its prediction leads to hit in OptPolicy. Depending on friendliness the RRIP value (indication of eviction , at a value of 7) is set to be 0 or 7. The paper trains the OPTgen to consider only cases where Prefetch helps a later demand access and ignores insertion for prefetches that will not be associated with demand access. However it doesn't completely ignore the prefetches to avoid memory congestion. The paper performs 4.5% IPC improvement over LRU without data prefetches whereas 2.25% with prefetches on single core [1]

2.3 Ship++ Replacement Policy

This paper is based on an earlier replacement policy called ship. The idea of the paper is to associate reuse characteristics of a cache line based on the signature of a line which can be a characteristic of PC and other meta data. This is done by choosing what RRPV value to be used on a cache hit/miss and associating corresponding confidence counter for signature. The paper tries to improve on the existing SHIP policy by suggesting multiple enhancements. First enhancement is to insert lines with RRPV =0 for high confidence (saturating) reuse of lines. Second is to weigh cache hits and misses similarly(update on first reference) and not have overtraining. Third enhancement associates RRPV = 3 (low priority) to writeback lines, since they are more likely to be evicted from here as well. Fourth is to use different signature for demand and prefetch requests and thus reduce interference. Fifth is the update criteria of prefetch requests if they are followed by prefetch or demand (less priority). Based on these enhancements the paper is able to achieve a 6.2% improvement in IPC over LRU for single core configuration without prefetcher and 4.8% with prefetcher [2].

3 Meta Data

3.1 Hawkeye

3.1.1 Meta Data There are several components of the replacement policy:

- Predictors - 2 different predictors each with 2K entries of 5 bit counter each to identify cache adverse and cache friendly lines taking 2.56KB space
- Sampler - It is a structure that is used to reduce the information required to construct OPT's behavior, this is a 2800 entry 4-byte wide table taking 11.2 KB space
- Occupancy Vector - This tracks the liveliness interval that overlap and hence identify the cache occupancy. This is a 128 entry 4-bit vector and each entry has 64 such vectors amounting to 4KB
- RRIP value per line which is a 3 bit per line of LLC taking 12KB space
- Information to identify Sampled sets which is a 64 set 16 way structure with 12 bits of information taking 1.5KB space .

3.1.2 Key Design - The most important policy of the paper are as follows :

- Identifying cache friendly and cache adverse entries by associating the behavior of OPTgen with the confidence counter. This allows to insert lines with an RRIP of 0 or 7 which will dictate how fast can they be evicted
- The second important aspect of the paper is to associate entries in the OPTgen for prefetch entries if they are followed by demand accesses. This is to associate less allocation of useless cache lines and make space for useful entries. However paper doesn't completely ignore redundant prefetches so as to avoid congestion.

3.2 Ship++

3.2.1 Meta Data - Over a baseline LRU consists of the following :

- Per line metadata in the LLC. This is 2 bits for the RRPV value and 1 bit to indicate if the line was added due to prefetch or not . The overhead of RRPV is 8KB
- Signature History Counter Table - This is a 16K entry 3bit counter that measures confidence of reuse amounting to 6KB for a core and 24KB for 4 cores
- Sampled Set for signature - This is a 64 set 16 way structure(1K entries) of 15-17 bits (14 - signature , 1- Reuse, 2- Only for multicore system) that tracks Reuse History of a signature. This is in total 1.875-2.125KB
- Storage to identify Sampled Set - This is a 64 Entry2byte wide table that identifies signatures for sampled set Table insertion. This is merely 128 Bytes in size.

3.2.2 Key Design - The most important policies of the Replacement policy are as follows :

- Insertion of lines with RRPV = 0 for saturated counter value. This is one of the key design ideas since the RRPV of 2 and 3 are inserted depending on the Confidence Counter. Giving a value of 0 keeps the block for farther time in the line thereby favouring the reuse characteristic
- The second is to bifurcate the re-reference behavior of demand accesses vs prefetcher accesses. With this it associates different confidence value for each access and leads to less interference between them, which is crucial in workloads that rely heavily on either prefetches or demand accesses only
- Writing Writebacks with RRPV of 3, this allows to prioritize Writeback evictions since the RRPV is already 3 and can safely be written into memory as they are not likely to be reused.

4 Simulation Methodology

The replacement policy is evaluated on the 3rd Cache Replacement Championship Infrastructure present at [4]. Traces are the same as ones run for the championship. In this paper we only check with the traces that are publicly available. There are total 200 available traces. However to reduce the runtime, 51 traces were selected from the pool of 200 traces based on their MPKI. Traces with significant MPKI was chosen by choosing a threshold MPKI. The policies are warmed up for 50M instructions and then run for 100M instructions for Hawkeye instead of 250M to speed up simulations /citehawkeye, warmup instruction count for Ship is 10M and 100M for running as mentioned in paper [2]. Results are then populated as seen in the graphs. We compare the performance of new replacement policies with Baseline LRU replacement policy. The IPC variation over all 51 workloads for the baseline of LRU, Hawkeye and Ship++ is represented in 1

The IPC Speedup attained from Hawkeye over Baseline is 1.029 as apposed to 1.0209 which is a margin of 0.7%, and MPKI reduction is 9.971%. The IPC Speedup attained from Ship++ is 1.02882 as apposed to 1.046 which is a margin of 4.4% and MPKI is found to be 8.506% .

5 Design Space Exploration

5.1 Hawkeye

The Parameter chosen for Design space exploration are the ways and number of sets from Sampler, and the occupancy vector size. Varying the associativity and number of sets is important to give a large range of the history information that can be stored without making false judgement. This is done without changing the size of the Sampler so as to not give unfair advantage of its usage. Occupancy vector size seems to be a second good choice for the exploration as it tracks what blocks have lived in a given cache block. Depending on what blocks are likely to remain more in this block, it gives a confidence to keep them and evict other entries.

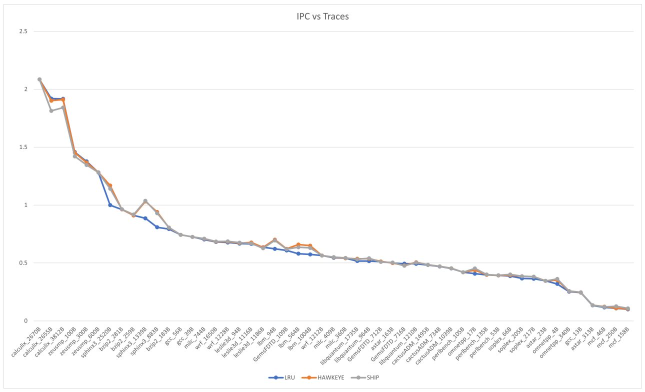

Figure 1: IPC of individual traces compared with LRU, Hawkeye and Ship Baseline versions

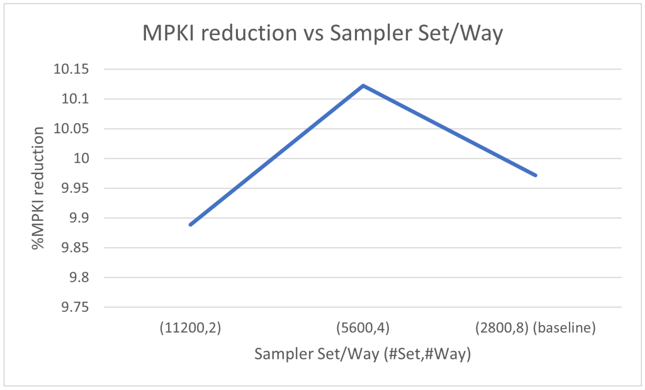

Figure 2: MPKI reduction with changing Hawkeye Sample Set size

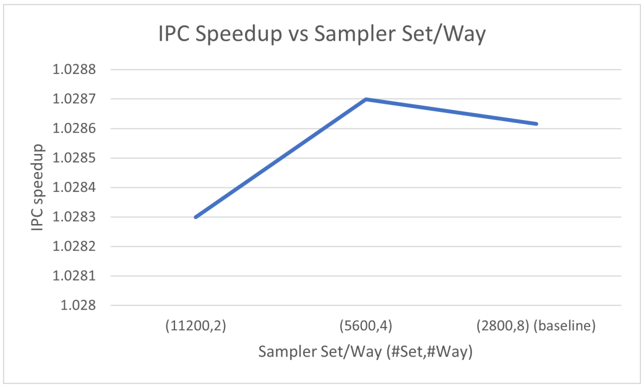

Figure 3: IPC Speedup with changing Hawkeye Sample set size

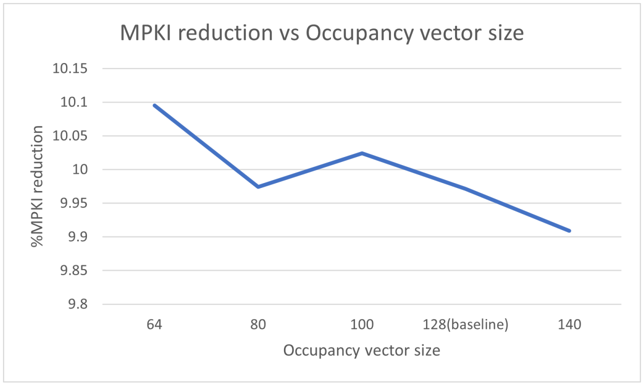

Figure 4: MPKI reduction with changing Hawkeye occupancy vector size

5.1.1 Varying associativity and sets of Sampler - In this exploration, we keep the size of Sampler to be same and vary the number of Sets and was to find an optimal setting where he replacement policy benefits. The Improvement can be seen in Figure 2 and 3. There are only 3 points of exploration here. The result for 1way has been omitted since it results into huge MPKI increase, which can be reasoned by the sampler having more conflicts in the cache then the it can benefit from the increased size. The ideal trend is to see the Baseline - 2800 Set and 8way associativity to have the best reduction in MPKI and IPC improvement. However it is seen at the 5600 set, 4way associative structure. This suggests that there are sufficient number of blocks that need to be identified but no more than 4 need to be accommodated in a block. having a 8 way structure limits the unique entries in 2 different sets which gives a wrong curve here. ANother possible reason is we have simulated the results for only 51 high MPKI traces, whereas paper has simulated for over 200 benchmarks as from the championship infrastructure. It is possible that the improvement from lower MPKI loads is significant and offsets the gains from a 4-way structure more than 8 way structure. If that is not the case, having a 4way structure seems more optimal at the same budget.

5.1.2 Varying occupancy vector size - Occupancy vector stores the access history of the cache over N number of blocks. It consists of 2 parts, one stores address and other stores sequence of occupancy. That gives a baseline of 128 entries, 64 for address and 64 for sequence. The ideal trend is to have optimal performance at baseline and increases from lower sides of the Baseline, but either saturates or very marginally increases beyond baseline. However the trend shows otherwise here as seen in Figure 4 and 5. The trend shows 64 entries to be performing better than 80 or 100 or 140 entries. This can be attributed to the fact that beyond 32 instances of sequence, the occupancy vector does not store any meaningful access history of blocks and it only leads to corruption of further predictions with higher history. A suboptimal configuration would have been 64 entries, but this isn't the case because we are only simulating the prefetcher configuration which works in conjunction with replacement policy. If we have additional results from single core system without prefetching and with higher number of workloads, we might get baseline to be at correct configuration.

5.2 Ship++

The Parameter chosen for Design space exploration are the max RRPV values and the number of Sampled sets. These are the two main parameters that can be varied without leading to huge change in the hardware budget. The second reason to choose such parameters is as follows. Having very little difference between maxRRPV value and 0 would evict

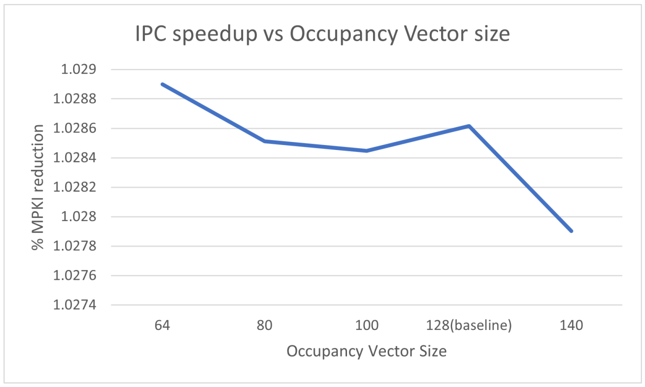

Figure 5: IPC Speedup with changing Hawkeye Occupancy vector size

even higher confidence entries faster, whereas more granular RRPV will place higher confidence sets for more interval in the cache. Since the replacement policy can't add meta data for each line in the LLC cache, it needs to sample certain number of sets and find high confidence entries among it, which makes it the second likely candidate for Design space exploration.

5.2.1 Varying maxRRPV value - In this exploration, we vary the maxRRPV value from 2 to 6, while for RRPV 4,5,6 the RRPV bitsize increases, the budget added due to that is still within the permissible limit of the championship. The trend is shown in Figure 6 and 7. The trend to expect would be good improvement at RRPV of 3/4 and then marginal improvements with higher RRPV. While this is seen in MPKI reduction, the IPC speedup shows otherwise with a dip. Carefully analyzing the dip shows the order of change between IPC improvement is in order of 0.01. While this graph seems a major dip, it is otherwise, nevertheless it makes sense to analyse what can be causing this if the MPKI is reducing. Higher MPKI reduction is seen in Traces WRF, ZEUSMP. These workloads are benefiting in reducing the misses but they still don't improve the IPC by a similar factor. This can be attributed to the fact that prefetches are helping these workloads at the L1 and L2 level resulting in lesser LLC misses, but the simulator is not accounting for the congestion for these workloads at the Memory controller level and gives the same latency to each access. And since Misses are averaged, certain high Miss regions might overshadow entire MPKI calculation and give false representation of where the misses are occuring and hence not give proper congestion idea.

5.2.2 Varying Sample set size - The number of sets being sampled are important as they contain the signature bits that will be used to index into the signature history Table. If we have lesser number of entries then we would not be able to reap full benefit of the Table size. Hence the ideal trend would be to expect increasing speedup and then saturating post optimal point. This can be seen in figure 8 and 9. IPC somehow gives a slightly correct trend barring the dip, the dip can be attributed to the fact that the Signature hash function might be related to a function that favours power of

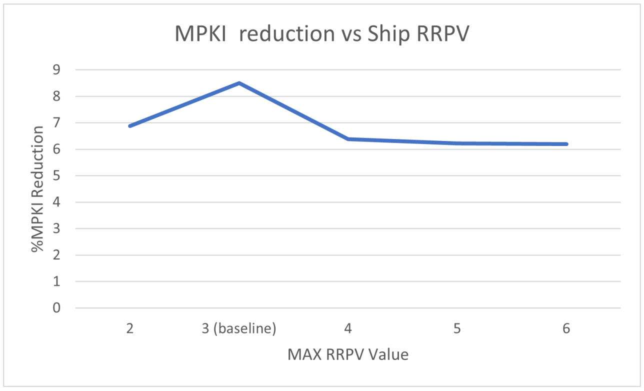

Figure 6: % MPKI reduction with varying max RRPV for SHIP policy

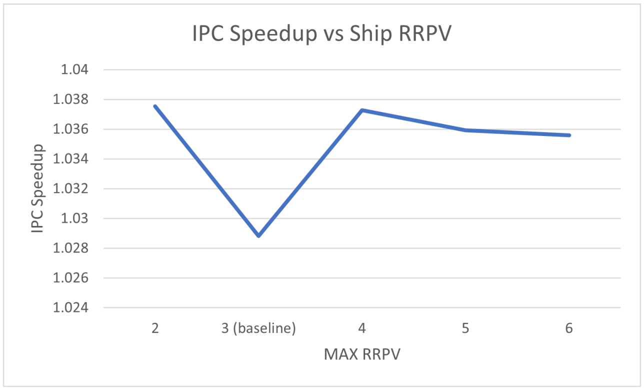

Figure 7: IPC Speedup with changing max RRPV for Ship policy

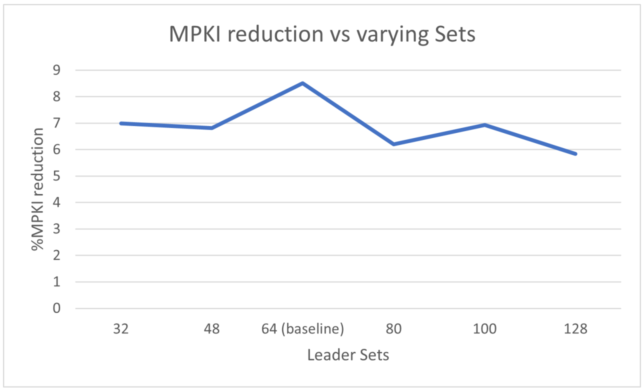

Figure 8: %MPKI reduction with varying Ship Sampling Set size

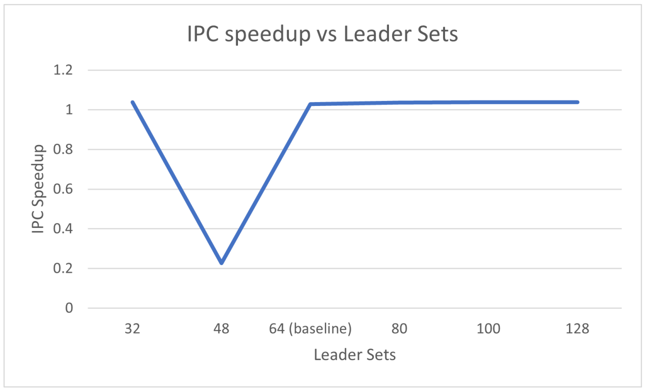

Figure 9: IPC Speedup with changing Ship Sampling Set size

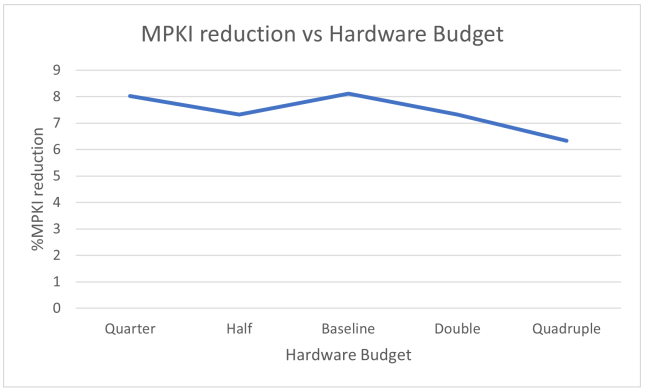

Figure 10: %MPKI reduction with varying hardware budget for Hawkeye

2 for proper accesses to have Bank interleaving. And already higher than 64 entries already reap the benefits of larger sampler size and doesn't result into mis-representation of signature accesses. The same can be attributed to the dips on 48 dip on the MPKI trend. The MPKI trend beyond 64 looks interesting. While it should have been stagnant /saturating, it seems decreasing. This might be due to the fact that The signature function may not be fully tuned for entries higher than 64 baseline and hence results into one to many mapping of signature to entry in the table, where the indexing is done on a first search basis. Consider the case that one signature is generated at Index I and second signature is generated at I2. With the way the signature gets generated, I2 might have been placed higher into the Table and may lead to false updates from next access onwards, whereas the actual update would have been done at Index I1 if I2 weren't present

6 Dealing with Hardware Budget

CRC stipulates a budget of 32KBytes of metadata per core. Here we have simulated the performance of the replacement policies at smaller and larger hardware budgets . Ideal expectation is to have increasing performance beyond the current limit upto a optimal value and then saturating or diminishing returns. Here we report only MPKI as the metric and ignore the IPC. Average MPKI is the number of Misses which the replacement policy is changing with respect to Baseline replacement policy. Secondly since this is not a cycle accurate simulator that tests the actual traffic, the IPC improvements may not give accurate results. However MPKI is still a good metric, because LLC is supposed to benefit less misses.

6.1 Hawkeye

MPKI improvements from first-order structures in the hawkeye replacement policy are seen in 10, the size of structures are stipulated from Quarter the budget all the way to Quadruple budget, in multiples of 2. For each configuration, the parameters that are varied are - Sampler entries and the RRPV information per line and the predictor size. One thing to note is, when we increase the predictor size, it automatically updates the signature length as well. The description of each hardware component and its estimated size and hardware utilization of budget is shown below. However the trend shows a reverse trend, indicating that with increasing the RRPV information, it takes longer time to train the confidence of classifying entries as cache friendly or Cache adverse in case of Hawkeye.

- 8 KB Budget- Sampler 700 entries - 2.8KB, predictors - 2 512 entry - 5bit counter tables (0.58KB) , 4-bit 32 entry 64 set Occupancy vector (1KB), RRPV - 1bit 32K entry(3KB) and Sample set 64set 16way 10bit per line (0.5KB). Actual utilization - 7.82KB
- 16 KB Budget- Sampler 1400 entries - 5.6KB, predictors - 2 1K entry - 5bit counter tables (2.56KB) , 4-bit 128 entry 64 set Occupancy vector (4KB), RRPV - 2bit 32K entry(6KB) and Sample set - 64set 16way 11bit per line (1KB). Actual utilization - 18.8KB
- 32 KB Budget- Sampler 2800 entries - 11.2KB, predictors - 2 2K entry - 5bit counter tables (2.56KB) , 4-bit 128 entry 64 set Occupancy vector (4KB), RRPV - 3bit 32K entry(12KB) and Sample set - 64set 16way 12bit per line (1.5KB). Actual utilization - 31.8KB
- 64 KB Budget- Sampler 5600 entries - 22.4KB, predictors - 2 4K entry - 5bit counter tables (5.32KB) , 4-bit 128 entry 64 set Occupancy vector (4KB), RRPV - 4bit 32K entry(24KB) and Sample set - 64set 16way 13bit per line (3KB). Actual utilization - 58.7KB
- 128 KB Budget- Sampler 11200 entries - 44.8KB, predictors - 2 8K entry - 5bit counter tables (10.64KB) , 4-bit 128 entry 64 set Occupancy vector (4KB), RRPV - 3bit 32K entry(48KB) and Sample set - 64set 16way 14bit per line (6KB). Actual utilization - 113.44KB

6.2 Ship++

MPKI reduction for Ship++ is shown in 11, this trend looks more realistic and shows that with increasing hardware budget the advantage is minimal. The size of structures are stipulated from Quarter the budget all the way to Quadruple budget, in multiples of 2. The parameter changes here is the RRPV bit width per line and Signature History Counter Table size. While it can be seen from the below hardware,

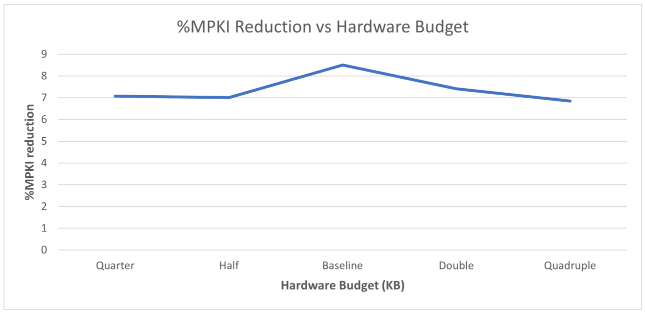

Figure 11: %MPKI reduction with varying Budget for Ship

the budget utilization of Ship++ is quite less and there can be more room for improvement. However, since its first order size don't linearly vary but rather vary exponentially, the parameter for optimal setting might need more tweaking. The trend is almost similar for each configuration, which means that most of the additional information by adding more number of Signature history is not required as it might already be enough around the optimal hardware budget. Secondly the signature function may not be fully optimal for higher budgets as described earlier and hence the graph shows a stagnant improvement after the baseline budget.

- 8KB budget - 2bit metadata per line - 6KB, 4K entry 3 bit counter - 1.5KB, Sample Set -64set - 1.875KB, Sample Set ID - 64 Entry - 128Bytes - 9.5KB Actual Utilization
- 16KB budget - 2bit metadata per line - 6KB, 8K entry 3 bit counter - 3KB, Sample Set -64set - 1.875KB, Sample Set ID - 64 Entry - 128Bytes - 11KB Actual Utilization
- 32KB budget - 3bit metadata per line - 12KB, 16K entry 3 bit counter - 6KB, Sample Set -64set - 1.875KB, Sample Set ID - 64 Entry - 128Bytes - 20KB Actual Utilization
- 64KB budget - 4bit metadata per line - 24KB, 32K entry 3 bit counter - 12KB, Sample Set -64set - 1.875KB, Sample Set ID - 64 Entry - 128Bytes - 38KB Actual Utilization
- 128KB budget - 5bit metadata per line - 48KB, 64K entry 3 bit counter - 24KB, Sample Set -64set - 1.875KB, Sample Set ID - 64 Entry - 128Bytes - 74KB Actual Utilization

6.3 Analysis against prefetcher

6.3.1 Hardware Cost Comparison

- L1I-Cache size for Instruction Prefetching championship was 32KBytes whereas LLC size for Cache Replacement Policy is 2MB per core and 8MB for multi-core system
- For the chosen Replacement Policies, the hardware budget utilization ranges from 20KBytes (Ship++) to 31.8KBytes (Hawkeye).
- Overhead of the above replacement Policy metadata with respect to LLC size is 0.97%(Ship++) and 1.56%(Hawkeye)
- Instruction prefetchers simulated in prior work were 96KBytes (FNL+MMA) and 125KBytes (D_JOLT) . The corresponding area overhead in comparison to L1I cache size is 300%(FNL+MMA) and 400%(D_JOLT)

6.3.2 Latency comparison - Instruction prefetchers are front end processors and need to work in advance than other units like Branch Predictor as well as Fetch Engine, hence they have to be computationally less expensive as they can not tolerate more access time, however they can tolerate more size for meta-data as the corresponding structures for which they are designed are relatively small. On the other hand, LLC sizes are quite big but their replacement policies can tolerate higher latencies as they are backend processes. Some of the reasons for computational latency in LLC structures vs Instruction prefetchers can be as follows:

- The associativity of structures like Occupancy Vector or Sample Set are higher than 16, somewhere even upto 64. Such highly associative structures take more time to access for comparison between different ways of a set. This can be one latency hungry computation. LLC cache lines typically being higher in size as well as ways require bigger comparators as well as cascaded muxes to resolve Hits.
- The LLC structures itself are huge and can take more time to access, plus they are dependent on demand misses and response from memory, whereas prefetchers are lightweight and run with minimal interaction from Branch predictors only to flush entries on mispredictions.
- Most of the replacement policies check for RRPV values, if RRPV of a cache line is not close to the max value it keeps on reiterating and adding the RRPV to find the first highest possible value which can incur latency in finding such RRPV.

7 Code

The code is available in repository : CSE240C HW2
Steps to run are added in File called StepsToRun.txt

8 References

[1] https://www.dropbox.com/s/dl/7riayq24gssxq1j/crc17-hawkeye.pdf

[2] https://www.dropbox.com/s/algx6mwxa591uoy/Ship%2B%2B.pdf

[3] https://www.dropbox.com/s/z685pmu1mn2lgr1/Expected%20Hit%20Count.pdf

[4] https://www.dropbox.com/s/o6ct9p7ekkxaoz4/ChampSi_CRC2_ver2.0.tar.gz

HawkShip - An adaptive replacement policy for Last Level Cache

Monil Shah A59012111

ABSTRACT

With increasing memory wall, it becomes crucial to improve the Cache Hits. Memory Latency can be hidden through multiple means. While we cannot have infinite size caches to avoid conflict misses, we can improve upon cache replacement policies to make sure conflicts are reduced. This results in better utilization of memory bandwidth by not thrashing the memory controller with accesses.Out-of-Order execution is another such paradigm however it would still reach a bottleneck in fetching new set Instructions. Prefetchers alleviate this problem by bringing the likely to be used Instruction/-Data Blocks before it is actually used. This paper presents an adaptive policy that choses between two replacement policies from the 3rd Cache Replacement Championship Participants and discussion about two Winner Prefetchers at the 1st Instruction Prefetching Championship based on the Simulator ChampSim.

1 Introduction

With growing instruction-working set among neural network, cloud and server applications, instruction foot print is ever increasing and so are cache misses. Instruction prefetchers effectively hide cache miss latency by prefetching the blocks speculatively. This is done in separate buffers so as to not affect the normal cache operation through cache pollution or conflict misses. If the data isn't used there is no additional penalty in comparison to not using prefetchers.To reduce cache thrashing, one of the ideas is to prefetch necessary data, whereas other is to have a better replacement policy. With better replacement policies it is possible to predict which blocks are likely to be reused later and evict entries that are less likely to be reused later. This paper is organized as follows : Section XXX provides high level summary of 2 winner prefetchers at the 1st Instruction Prefetching Championship and 2 winner replacement policies at 3rd Cache Replacement Championship, Section XXX lists the motivation of this study behind combining cache replacement policies, Section XXX Provides simulation results for Baseline Policies and winner policies. Section XXX introduces architecture of the adaptive replacement policy, Section XXX is conclusion followed by Section 7 which contains the Code and Scripts involved in running these simulations

2 Literature

Here we present the replacement policies and the instruction prefetchers that we compare against.

2.1 LRU Replacement Policy

LRU is a standard cache replacement policy that keeps track of cache lines that were used farthest in past than other cache lines, and aims to remove them assuming they are not likely to be re-referenced in the future. LRU inserts a line at Most Recently used position and it takes the cache line to go from all the way from MRU to LRU position to be evicted. The data structure can be considered as a double linked list, where on every hit the cache line is brought closer to the Head of MRU side and the least recently used is placed at the farther side. The problem with such a scheme is the number of bits required to represent the LRU structure is n! where n is the number of ways of a set, which grows quite fast. There are other variants of LRU like psuedo LRU, tree LRU that use less number of bits to represent the structure.

2.2 Hawkeye++ Replacement Policy

This paper is based on modifying the existing Hawkeye replacement policy. Earlier policy treats demand and prefetch requests in same way, but the performance impact of both will be different depending on the workload, so to optimally decide on the replacement, the paper uses separate tables for demand and prefetch requests. The paper tries to construct Belady's algorithm by associating Load instruction with being cache friendly or cache adverse. Friendly lines are given higher priority to inert into the cache. By associating the liveness of blocks in cache in the past with future access, it determines what should be the occupancy of the blocks in the future. The Hawkeye predictor identifies the lines to be friendly or adverse depending on confidence counter. It works in correlation with Optgen to strengthen the confidence in previous PC if its prediction leads to hit in OptPolicy. Depending on friendliness the RRIP value (indication of eviction , at a value of 7) is set to be 0 or 7. The paper trains the OPTgen to consider only cases where Prefetch helps a later demand access and ignores insertion for prefetches that will not be associated with demand access. However it doesn't completely ignore the prefetches to avoid memory congestion. The paper performs 4.5% IPC improvement over LRU without data prefetches whereas 2.25% with prefetches on single core [1]

2.2.1 Meta Data There are several components of the replacement policy:

- Predictors - 2 different predictors each with 2K entries of 5 bit counter each to identify cache adverse and cache friendly lines taking 2.56KB space
- Sampler - It is a structure that is used to reduce the information required to construct OPT's behavior, this is a 2800 entry 4-byte wide table taking 11.2 KB space
- Occupancy Vector - This tracks the liveliness interval that overlap and hence identify the cache occupancy. This is a 128 entry 4-bit vector and each entry has 64 such vectors amounting to 4KB

- RRIP value per line which is a 3 bit per line of LLC taking 12KB space
- Information to identify Sampled sets which is a 64 set 16 way structure with 12 bits of information taking 1.5KB space .

2.2.2 Key Design - The most important policy of the paper are as follows :

- Identifying cache friendly and cache adverse entries by associating the behavior of OPTgen with the confidence counter. This allows to insert lines with an RRIP of 0 or 7 which will dictate how fast can they be evicted
- The second important aspect of the paper is to associate entries in the OPTgen for prefetch entries if they are followed by demand accesses. This is to associate less allocation of useless cache lines and make space for useful entries. However paper doesn't completely ignore redundant prefetches so as to avoid congestion.

2.3 Ship++ Replacement Policy

This paper is based on an earlier replacement policy called ship. The idea of the paper is to associate reuse characteristics of a cache line based on the signature of a line which can be a characteristic of PC and other meta data. This is done by choosing what RRPV value to be used on a cache hit/miss and associating corresponding confidence counter for signature. The paper tries to improve on the existing SHIP policy by suggesting multiple enhancements. First enhancement is to insert lines with RRPV =0 for high confidence (saturating) reuse of lines. Second is to weigh cache hits and misses similarly(update on first reference) and not have overtraining. Third enhancement associates RRPV = 3 (low priority) to writeback lines, since they are more likely to be evicted from here as well. Fourth is to use different signature for demand and prefetch requests and thus reduce interference. Fifth is the update criteria of prefetch requests if they are followed by prefetch or demand (less priority). Based on these enhancements the paper is able to achieve a 6.2% improvement in IPC over LRU for single core configuration without prefetcher and 4.8% with prefetcher [2].

2.3.1 Meta Data - Over a baseline LRU, it consists of the following :

- Per line metadata in the LLC. This is 2 bits for the RRPV value and 1 bit to indicate if the line was added due to prefetch or not . The overhead of RRPV is 8KB
- Signature History Counter Table - This is a 16K entry 3bit counter that measures confidence of reuse amounting to 6KB for a core and 24KB for 4 cores
- Sampled Set for signature - This is a 64 set 16 way structure(1K entries) of 15-17 bits (14 - signature , 1- Reuse, 2- Only for multicore system) that tracks Reuse History of a signature. This is in total 1.875-2.125KB
- Storage to identify Sampled Set - This is a 64 Entry2byte wide table that identifies signatures for sampled set Table insertion. This is merely 128 Bytes in size.

2.3.2 Key Design - The most important policies of the Replacement policy are as follows :

- Insertion of lines with RRPV = 0 for saturated counter value. This is one of the key design ideas since the RRPV of 2 and 3 are inserted depending on the Confidence Counter. Giving a value of 0 keeps the block for farther time in the line thereby favouring the reuse characteristic
- The second is to bifurcate the re-reference behavior of demand accesses vs prefetcher accesses. With this it associates different confidence value for each access and leads to less interference between them, which is crucial in workloads that rely heavily on either prefetches or demand accesses only
- Writing Writebacks with RRPV of 3, this allows to prioritize Writeback evictions since the RRPV is already 3 and can safely be written into memory as they are not likely to be reused.

2.4 Next line Prefetching

Next line Prefetcher is a basic form of prefetcher which tries to exploit spatial locality. The idea behind this is, if a block is used, a consecutive memory location block is more likely to be used unless there are branch instructions of procedure calls or jumps. Since the branches and procedure calls are not a huge factor of the instruction footprint, fetching few blocks along with a miss helps in reducing stalls.

2.5 D_JOLT Prefetcher

D-JOLT predictor [5] is a return address stack based prefetcher. The return address stack is used to predict the return address of the next set of instructions. During the learning phase it generates a signature based on the Hash of Return Addresses in the Stack and associate with corresponding cache miss that was observed. It is based on a RDIP prefetcher with the only difference being, the Stack of Return Addresses uses all addresses instead of evicting them along with a counter to distinguish the signature, this helps in adding some correlation between previous function calls. D-Jolt uses a total of 3 prefetchers, each with its own unique property to compensate for the accuracy/ timeliness drops of other prefetchers. The long and short prefetchers use tables to record signatures and miss addresses. Fallback prefetcher is a stream based prefetcher [9].

2.4.1 Meta Data - The D_JOLT prefetcher uses 3 prefetchers. The long range prefetcher has a signature queue consisting 15 deep 23-bit wide entries. The signature Generator consists of a 7deep 32-bit wide FiFo and a 32-bit Counter. The miss table consists of 2048 set, 4way associative deep 76-bit(Tag, LRU, miss vector) wide entries. The short range prefetcher has a signature queue consisting 4 deep 23-bit wide entries. The signature Generator consists of a 4 deep 32-bit wide FiFo and a 32-bit Counter. The miss table consists of 1024 set, 4way associative deep 77-bit (Tag, LRU, miss vector) wide entries. The fallback prefetcher uses 16 deep 65-bit wide (Tag, replacement, address) entries Train table and 16 deep 63-bit (Tag, replacement, address) wide entry Monitor Table. There is a upper bit table consisting of

31 entries and 41 bit wide (Tag, Valid). There is a miss table of 256 set, 4 way 79 -bit wide (Tag, LRU, miss vector). All totalling to 125KBytes

2.4.2 Key Design - The key design of the prefetcher is the signature generator, the miss tables and the signature queue. The fifo stores return addresses from the stack and implements a counter that counts number of returns to associate a unique signature instead of limited signature.. Long range prefetcher has 7 deep FIFO and short range has 4 deep. The signature queue implements the distance parameter. By adding the signatures in a queue of N-Deep, the prefetcher associates a miss address on popping a signature from queue, which would be N-deep in the past. Miss table associates signature with Miss address. Upper bit table is added as a fully associative cache to reduce size of normal miss table.

2.6 FNL+MMA Prefetcher

FNL-MMA Prefetcher [6] uses a combination of two prefetchers. The FNL prefetcher is based on the idea that all next lines should not be prefetched, as it causes a lot of L2 accesses and over-pollution. Instead only meaningful and reasonable to be used lines should be prefetched. By associating the current miss with previous misses and a 2bit counter, the next N blocks are fetched. The corresponding next miss need not prefetch again and uses a filter mechanism to reduce prefetches. The MMA prefetcher prefetches non-contiguous lines and tries to foresee several blocks ahead in what instruction blocks are likely to be missed based on the current misses. By associating confidence with effect of one miss on another, it can look ahead into misses.

2.5.1 Meta Data There are several key components. I-shadow cache with 192 entry 17-bit wide Table that is similar to a Icache used to trigger prefetch. It uses a 64K entry 1-bit Touched and 2-bit WorthPF Tables. These are used to predict which blocks to fetch. FNL prefetcher uses a 128 entry 17 bit FNL filter to skip requesting already present blocks in prefetch buffer. MMA prefetcher uses an 8K entry Miss table which is 71 bit wide (tag, block-address, control bits) and a 24 entry 58bit wide MMA filter. Total hardware budget lies around 96KBytes. And FNL filter reset interval of 8K. The FNL Prefetcher uses a distance of 5 blocks and MMA uses a distance of 9 blocks. There is a slight difference in the paper submitted to the conference vs the actual code used. The IPC speedup achieved with Old parameters (9 distance and 16 entry MMA filter) is 1.28745 whereas with the updated code (11 MMA distance and 24 filter) is 1.292. We will refer the FNL+MMA in this paper as the Prefetcher with the updated parameters

2.5.2 Key Design - The touched Entry table is a 1bit counter of recently touched block flag along with a counter to indicate its demand access in a dedicated fix interval. The Touched Entry is set when a miss occurs, and the corresponding counter of previous Block is set to 3, to indicate that a miss on B causes miss on B' . By changing the confidence counter of miss intervals, it correlates when to prefetch and when not to. Next time when a Block Miss occurs, depending on the Counter, next 5 blocks are prefetched. The Miss interval dictates the time to refresh flags and clear Recentness, to only fetch meaningful data. The MMA Prefetch Table associates the miss address with predicted address using a Cache like table of Tags and Block Address. The MMA prefetcher prefetches 9 blocks in advance since we need to prefetch blocks well in advance. This is slightly different from a Next predicted Miss prefetcher as it can look N blocks ahead of potential miss.

3 Motivation

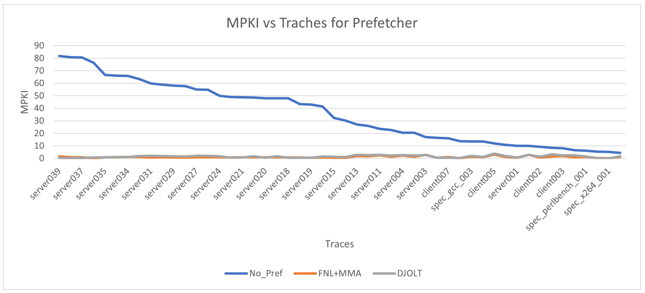

Figure 1: MPKI of individual traces compared with No_Prefetcher, FNL+MMA and D_JOLT

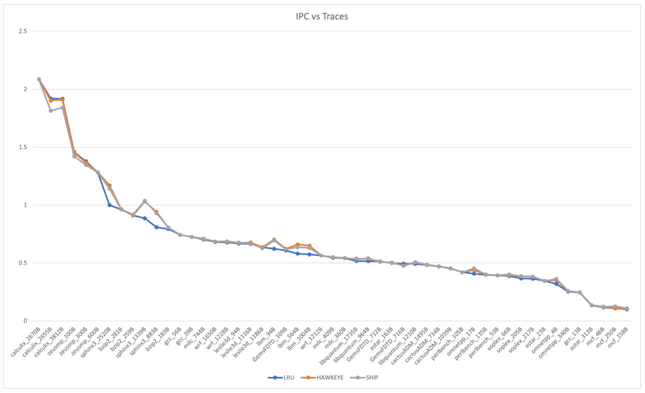

Figure 2: IPC of individual traces compared with LRU, Hawkeye and Ship Baseline versions

Fig 1 shows the improvement in MPKI for the first prefetching championship traces on using prefetcher on L1 instruction cache. The comparison is done against FNL_MMA and D_JOLT with a no prefetcher case. It can be clearly seen that the improvement is massive on adding a prefetcher. This is because the likely to be used lines are fetched before they are used, hence incurring fewer misses.

Fig 2 shows the improvement in IPC for the third replacement championship traces on using LLC replacement policies on shared Last level Cache. The comparison is done against Ship++ and Hawkeye++ with a LRU case. It can be seen that the improvement is not so massive. This is because Last level caches are big but there is not enough budget to have metadata corresponding to all the entries, hence the idea of sampling is used to generalize the idea of replacement across LLC. While it may not be optimal, it still reduces he misses and improves IPC

Some studies [8] have shown that not all cache lines need to be given an MRU position for an LRU scheme, because it occupies space in the cache before it can be evicted, if it was anyway not going to be re-referenced before eviction. Modern cache replacement policies try to use this to the advantage to track dead cache lines that are likely to be evicted before being re-referenced and place them in a LRU position instead of MRU position, while this does not reduce the miss in evicting the dead block, it gives performance boost for subsequent accesses to other cache lines that are likely to be re-referenced.

Replacement policies need a metadata structure for tracking such scenarios. While smaller caches can have one to one mapping between cache line and the replacement policy metadata, but larger caches cannot have one to one mapping between cache data and the metadata. Hence replacement policies use clever techniques to go past this issue. Most of the policies implement a counter (RRPV) [7] that mimics how many times the cache line was re-referenced before evicting it. The idea is that lines can be inserted with RRPV of 0 to max, and max RRPV lines should be evicted first. So RRPV becomes an ageing factor. Typically 3 bits suffice for tracking the interval for most of the cases.

Just like McFarling [10] predictor used a local and gshare predictor to beat a tournament predictor, it might seem intuitive to combine two prefetchers or two replacement policies and get a compounded gain. The hardware budget used in prefetching championship was 128KB, so to have meaningful performance over the existing two prefetchers, similar gains need to be shown at the same hardware budget. Main components used in the prefetchers are the signature generators and the tables to store confidence. Prefetchers hold more of spatial locality then temporal locality, so it is not easy to generalize the behavior of one prefetcher at a lower budget, since two events different in space need to be represented by different signatures. However, for LLC replacement policies it is different, since the policies rely on behavior of a subset of LLC sets to generalize the behavior of the LLC, it is possible to duel two replacement policies. The reasoning for that is, even though the generalization using sets is working, it can predict certain workloads better than all workloads since different workloads leave a different cache footprint. And due to a smaller set, it is very likely for policies to work better on different policies. If we have a way to combine two policies without destructive interference, we can achieve even better performance.

[8] introduces us to 4 different ways of adapting replacement policies

3.1 Static Insertion

In this type of insertion, all the lines can be placed in LRU position for LRU or with maxRRPV for RRPV based policies, only if the lines are referenced again, should the lines be moved to MRU for LRU or 0 RRPV for RRPV based policies, this makes the case of handling dead blocks easily. These blocks will be the first candidates to be evicted and there won't be a need for Incremental checking of RRPV for RRPV based policies.

3.2 Bimodal Insertion

This is a slight variation to the Static insertion policy, instead of placing all the lines in LRU or max RRPV, lines should be placed in MRU or 0 RRPV positions based on some probability threshold. This probability can be implemented using simple saturating counters ather than complex algorithms. ON every Hit the counter is decremented towards 0 and incremented on misses. If there are more hits then the counter saturates towards 0 and will allow to insert lines at MRU position or 0 RRPV indicating that these can be cache friendly lines.

3.3 Dynamic Insertion

While static policy is easier to implement, it does not adapt to running workload. It is found that certain workloads benefit from a certain policy and it would be beneficial if cache line friendly replacement policy dictate the insertion and eviction. Dynamic insertion allows to choose between 2 different policies depending on which one is incurring fewer misses. Each policy can keep metadata corresponding to each set and dependin on a saturating counter the policy that is currently incuring lower misses can be chosen.

3.4 Set Dueling

The higher in hierarchy the caches, the bigger the structures needed to track the cachelines. So to minimize the metadata structures, we require to shrink the structure. This is where set dueling helps, it dedicates certain number of sets to one policy, certain number of sets to other policy and remaining sets are decided based upon a saturating counter. This counter is brought towards one policy on misses in other policy's prediction. By keeping just a saturating counter and small structure to track which sets are dedicated to a policy, the overhead of this policy is least. This also augments from the fact that LLC replacement policies are generalized using sample sets which can closely represent the behavior of the whole cache.

4 Architecture

The storage budget of Hawkeye structures amounts to 31.2KB and that of ship amounts to 20KB, in order for the adaptive policy to have the same or better improvement, the budget should still fit within the union of the individual hardware budgets. However it is noticed that most of the structures across Hawkeye and Ship are not same, the only similar thing between Hawkeye and Ship is the RRPV value. Hawkeye uses a static search of maxRRPV or the max among the current RRPV values and uses an aging factor to account for not keeping stale RRPV values. Whereas Ship uses an iterative increment approach to find the first set that has maxRRPV values, this itself can be termed as the aging factor. Hence we compare the performance of a system with simple inclusion of bot structures and combining RRPV of both structures. Combining RRPV gives least performance degradation so we go ahead with that approach suggesting that both Hawkeye and Ship are quite orthogonal in their sampling, but their metadata uses similar approach to update. To further reduce the budget, we compare whether having a RRPV value of 3 suffice or 7. It is seen that most of the performance gains are still intact when moving to a lower RRPV, indicating that the workloads have a re-reference interval lesser than the cache ways.

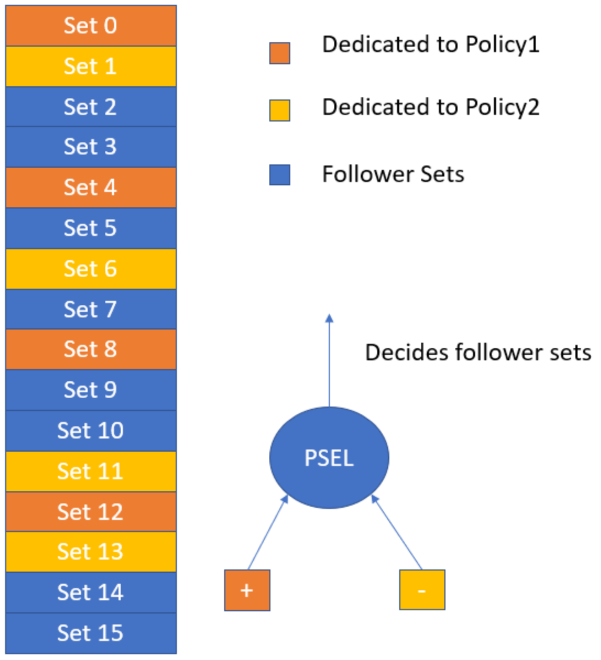

Figure 3: Dynamic insertion using set dueling

With the current structure the hybrid design HawkShip is still not within the hardware budget bound. So to analyze for further optimizations the following structures are resizedor combined:

- Predictor Tables reduced from 2K to 1K entry each, further reducing the signature size
- Signature entries for Ship reduced from 16K to 8K
- Signatures of Hawkeye and Ship reduced by 1 bit each
- Common 2-bit RRPV

All these optimizations bring the hardware budget into 32KB. Thus our baseline design for Adaptive policy becomes the following :

Hardware Budget

Component	Parameter	Budget
Hawkeye Sampler	2800 4-byte entries	11.2KB
RRPV	2 bit per line	8KB
Hawkeye Predictor	2 1K entry 4-bit Cntr	1KB
Occupancy Vector	64 vectors, 64 4-bit entry	2KB
is_prefetched	1-bit per line	4KB
Ship Samples		1.87KB
Ship Sample Set ID		128B
Ship SHCT	16K entry per core	3KB
Hawkeye Sample State	64 set 16 way 11-bit	1.2KB
Set dueling Counter		2B

This totals to a 29.88 KB budget. Next we present sensitivity study of the given design over multiple configurations.

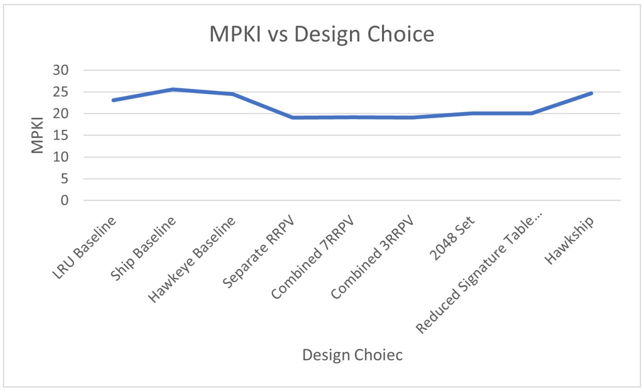

Figure 4: MPKI variation with Prefetcher configuration across different Design states

5 Simulation Methodology

The replacement policy is evaluated on the 3rd Cache Replacement Championship Infrastructure present at [4]. Traces are the same as ones run for the championship. In this paper we only check with the traces that are publicly available. There are total 200 available traces. However to reduce the runtime, 51 traces were selected from the pool of 200 traces based on their MPKI. Traces with significant MPKI was chosen by choosing a threshold MPKI. The policies are warmed up for 50M instructions and then run for 100M instructions for Hawkeye instead of 250M to speed up simulations [1], warmup instruction count for Ship is 10M and 100M for running as mentioned in paper [2]. Results are then populated as seen in the graphs. We compare the performance of new replacement policies with Baseline LRU replacement policy. The IPC variation over all 51 workloads for the baseline of LRU, Hawkeye and Ship++ is represented in 1 Adding over previous work, both configurations of single core were simulated i.e. with and without prefetcher.

Speedup for Replacement Policies with Prefetcher

Configuration	Design	IPC Speedup	MPKI
LLC with Prefetcher	LRU	1	23.055
LLC with Prefetcher	Ship	1.02882	25.573
LLC with Prefetcher	Hawkeye	1.029	24.54
LLC with Prefetcher	Hawkship	1.0223	24.679

Speedup for Replacement Policies without Prefetcher

Configuration	Design	IPC Speedup	MPKI
LLC no Prefetcher	LRU	1	17.86
LLC no Prefetcher	Ship	1.05	18.73
LLC no Prefetcher	Hawkeye	1.047	18.58
LLC no Prefetcher	Hawkship	1.044	19.19

6 Design Space Analysis

The path to reach the optimal design choice involved multiple attempts

6.1 Combining Policies

To start the exploration, both policies were simply merged without changing or modifying their structures. This was

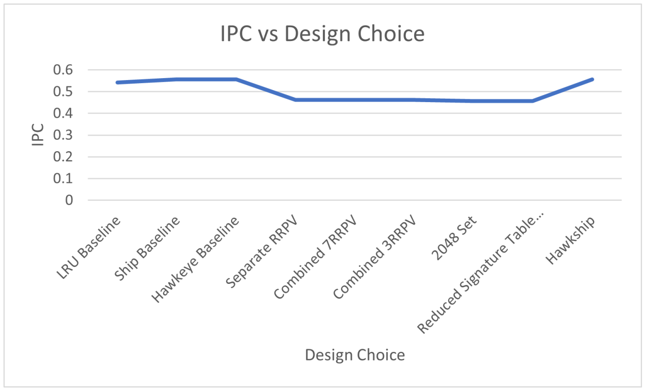

Figure 5: IPC variation with Prefetcher configuration across different Design states

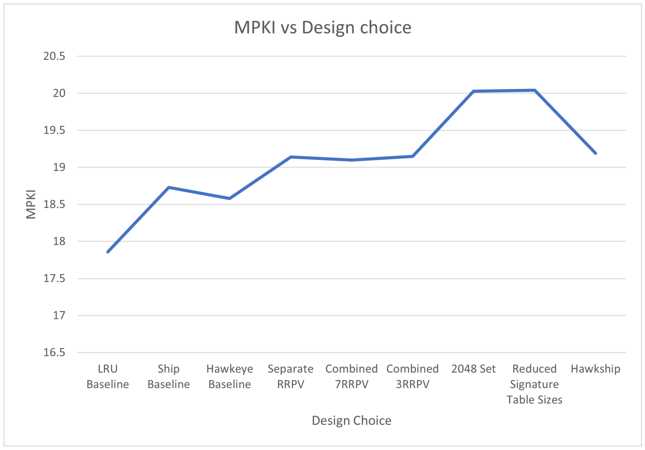

Figure 6: MPKI variation with no Prefetcher configuration across different Design states

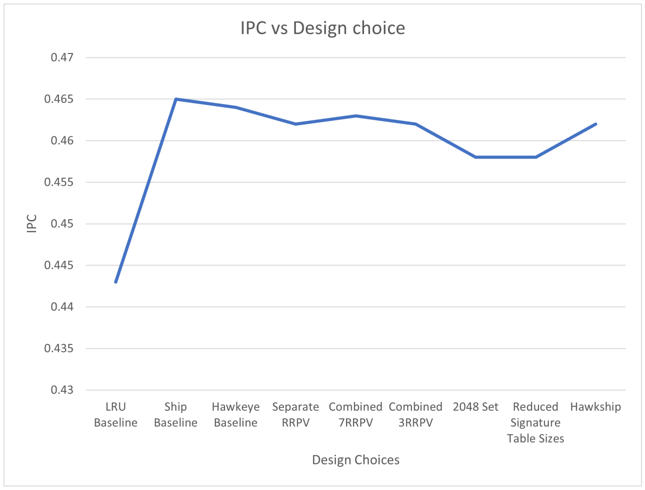

Figure 7: IPC variation with no Prefetcher configuration across different Design states

to check if there is a huge destructive interference between the two policies. Care was taken to rename the defines and other structures like SHCT and RRPV so that the simulation isn't modelled wrongly. The observation from figures 3 to 6 for points "Ship Baseline" , " Hawkeye Baseline" and "Separate RRPV" can be seen and shows that it falls as per with the expectation that set dueling can be implemented without major interference.

6.2 Combining RRPV

In this exploration, to minimize the hardware budget, the RRPV structures are combined. While ship does an iterative increment of RRPV if it doesn't find a maxRRPV line, Hawkeye simply evicts entry that has the current maximum among the lines. At a first glance this seems to be orthogonal and not combine-able, however, even Hawkeye does an ageing of the RRPV values during replacement update state. So as can be seen from figures 3 to 6, the MPKI and IPC of combined RRPV is not very bad. Also to compact the size further the RRPV was reduced from 7 to 3, and no noticeable change was observed. This shows that the structures have different ways of aging the RRPV value but the final effect is similar. The structures don't fail terribly on reducing the RRPV value from the points "Combined 7RRPV" and "Combined 3RRPV" indicating that the maximum re-reference interval is well within the number of ways in a cache.

6.3 Reducing Hawkeye Sets

Here, the sampler entries of Hawkeye were reduced from current 2800 to 2048 and all the way down to 1024 (not shown), the results gathered from this exploration revealed that 2800 is an optimum sampler size and anything below it simply results into higher MPKI and lower IPC. It an be reasoned by the sampler having more conflicts in the cache then the it can benefit from the reduced size. Thus the sampler entries for hawkeye are kept to be 2800. This can be seen from the points "Combined 3 RRPV" and "2048 set" in figures 3 to 6

6.4 Reducing Signature Tables and Counters

Signature tables associate the events of sampling by the replacement policy to reduce aliasing. It gives a notion of different events and their associated confidence counters, to help decide with certainty whether a line should be added with maxRRPV value or 0, or whether it is to be ignored for prefetch cases and writeback. By changing the signature sizes from "2048Set" to "Reduced Signature Table" in figures 3 to 6, no significant performance degradation is seen. This can be attributed to the fact that the number of unique events that need to be hashed by the policy is sufficient in lesser signature table.

7 Analysis

The structures of individual policies is as follows. Ship - 20KB, Hawkeye - 31.88 KB , Hawkship - 29.88KB. The IPC speedup for both configurations of Hawkship with and without prefetcher is still less than Hawkeye and Ship even though we have tried to combine two replacement policies. several observations can be made on the design exploration

- Ship is already optimal at 20KB budget using its structures, anything on top of it is adding interference, even

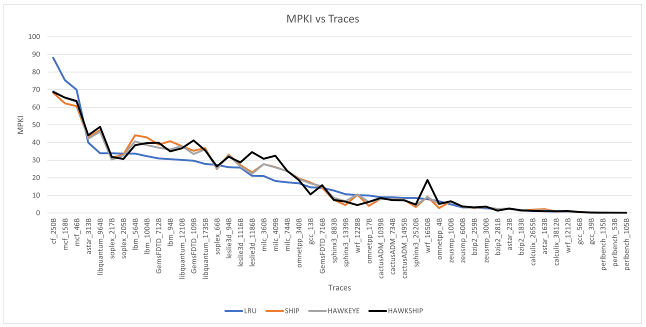

Figure 8: MPKI variation with no Prefetcher configuration

though the difference in IPC is minimal. For configuration with prefetcher the IPC of Hawkship is 1.0223 which is roughly 0.63 % (ship) and 0.65% (hawkeye), for configuration without prefetcher the IPC of Hawkship is 0.57% less from ship and 0.286% less from hawkeye

- From Figures 8 and 9 it can be seen that MPKI for Ship and Hawkeye is sometimes worse than LRU and same is observed from Hawkship as well, indicating that for some workloads the Reeference Interval mechanism incorrect as it does not correctly model LRU behavior. Or it can also be attributed to the fact that while the lines are inserted in LRU, they get referenced, but the optimization of Ship and Hawkeye to insert with RRPV 7 causes them to be evicted early and hence no re-reference, thereby increasing misses.
- There are some workloads where Hawkship is performing better than Hawkeye and Ship like sophinx,omnetpp mcf, so clearly there is some scope if not zero for set dueling.
- There is huge magnitude drop in IPC across Prefetching and Non prefeching cases even for the baseline configurations of Ship and Hawkeye, which indicates that their existing policy of identifying prefetch requests and associating optimized RRPV is not fully optimal. The second reason can be that prefetches take unnecessary space in case of Inclusive caches in the LLC and thus it causes less available space for useful sets to remain in the cache.
- For the current configuration and design space exploration, it seems Ship and Hawkeye are already performing better and are quite similar and thus it is not a right candidate for set dueling. The idea of set dueling is to have orthogonal policies that work better for certain workloads. Since the MPKI variation is minimal between Ship and Hawkeye from 2 there is not much scope for set dueling Adaptive policy

8 Prefetcher vs Replacement

This section tries to compare the speedup obtained by using either the Instruction Prefetchers or Cache Replacement Policies over their baseline approach.

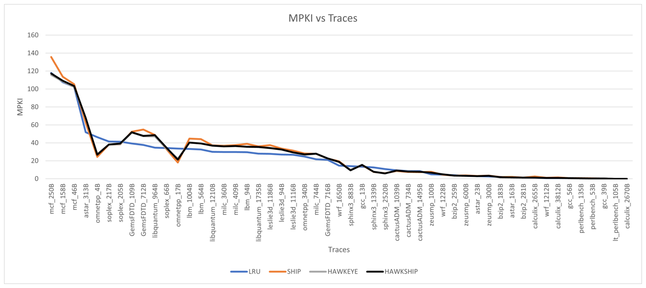

Figure 9: MPKI variation with Prefetcher configuration

8.1 Speed up analysis

- L1I-Cache size for Instruction Prefetching championship was 32KBytes whereas LLC size for Cache Replacement Policy is 2MB per core and 8MB for multi-core system
- Overhead of Ship metadata at 20KBytes for the Prefetch version is 0.97% with respect to LLC size. The overhead of Hawkeye policy at 31.8KBytes is 1.56% with respect to LLC size.
- In previous section it was seen that the speedup of Ship is 2.882% for configuration with prefetcher and that of Hawkeye is 2.9% over the LRU baseline. With the structure sizes that they have, the speedup per KB of both these structures is 0.1441 (ship) and 0.09 (Hawkeye). Clearly Ship performs better than Hawkeye at lower budget and hence higher speedup per KB
- Instruction prefetchers simulated in prior work were 96KBytes (FNL+MMA) and 125KBytes (D_JOLT) . The corresponding area overhead in comparison to L1I cache size is 300%(FNL+MMA) and 400%(D_JOLT)
- In previous paper it was seen that the speedup of D_JOLT is 28.94 and that of FNL+MMA is 28.7 over the no prefetcher baseline. With the structure sizes that they have, the speedup per KB of both these structures is 0.2315 (D_JOLT) and 0.302 (FNL+MMA). Clearly FNL+MMA performs better than D_JOLT at lower budget and hence higher speedup per KB
- Comparing LRU against other replacement policies vs comparing No prefetcher against prefetchers doesn't give a good comparison. So speedup of Prefetchers as compared against next line was computed and it is 1.277 for D_JOLT and 1.27584 for FNL+MMA. With the structure sizes that they have, the speedup per KB of both these structures is 0.2216 (D_JOLT) and 0.289 (FNL+MMA). Clearly this speedup per KB is lower than when compared with no-prefetcher but still better than replacement policies.

Given a resource constraint in the hardware budget, it would make more sense to invest the budget for a better prefetching policy (frontend processer) than for replacement policy(backend processor).

8.2 Latency comparison

Instruction prefetchers are front end processors and need to work in advance than other units like Branch Predictor as well as Fetch Engine, hence they have to be computationally less expensive as they can not tolerate more access time, however they can tolerate more size for meta-data as the corresponding structures for which they are designed are relatively small. On the other hand, LLC sizes are quite big but their replacement policies can tolerate higher latencies as they are backend processes. Some of the reasons for computational latency in LLC structures vs Instruction prefetchers can be as follows:

- The associativity of structures like Occupancy Vector or Sample Set are higher than 16, somewhere even upto 64. Such highly associative structures take more time to access for comparison between different ways of a set. This can be one latency hungry computation. LLC cache lines typically being higher in size as well as ways require bigger comparators as well as cascaded muxes to resolve Hits.
- The LLC structures itself are huge and can take more time to access, plus they are dependent on demand misses and response from memory, whereas prefetchers are lightweight and run with minimal interaction from Branch predictors only to flush entries on mispredictions.
- Most of the replacement policies check for RRPV values, if RRPV of a cache line is not close to the max value it keeps on reiterating and adding the RRPV to find the first highest possible value which can incur latency in finding such RRPV.

9 Acknowledgement

This report wouldn't have been possible without constant support from Professor Samira Mirbagher, who has constantly motivated and guided on the methodology of this study. Zhenya Ma for helping setup the infrastructure, colleagues like Arpan, Edwin, Tanmay, Brandon, Gandhar with whom I have had multiple rounds of discussion and strategy to optimize this study

10 Code

The code is available in repository : CSE240C HW3
Steps to run are added in File called StepsToRun.txt

11 References

[1] https://www.dropbox.com/s/dl/7riayq24gssxq1j/crc17-hawkeye.pdf

[2] https://www.dropbox.com/s/algx6mwxa591uoy/Ship%2B%2B.pdf

[3] https://www.dropbox.com/s/z685pmu1mn2lgr1/Expected%20Hit%20Count.pdf

[4] https://www.dropbox.com/s/o6ct9p7ekkxaoz4/ChampSi_CRC2_ver2.0.tar.gz

[5] https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/D-JOLT.pdf

[6] https://research.ece.ncsu.edu/ipc/wp-content/uploads/2020/05/FNLMMA-final.pdf

[7] https://dl.acm.org/doi/10.1145/1815961.1815971

[8] https://people.csail.mit.edu/emer/papers/2007.06.isca.dip.pdf

[9] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in HPCA, 2007, pp. 63–74

[10] https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf