INSTRUCTION PREFETCHERS AND CACHE REPLACEMENT POLICIES

An Undergraduate Research Scholars Thesis

by

ALEX CHRISTIAN[1] AND DAVID CHAPA[2]

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Paul V. Gratz

May 2021

Major: Computer & Electrical Engineering[1]
Computer & Electrical Engineering[2]

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

We, Alex Christian[1] and David Chapa[2], certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Instruction Prefetchers and Cache Replacement Policies

Alex Christian[1] and David Chapa[2]
Department of Electrical and Computer Engineering[1]
Department of Electrical and Computer Engineering[2]
Texas A&M University

Research Faculty Advisor: Paul V. Gratz
Department of Electrical and Computer Engineering
Texas A&M University

Processing speeds are determined by how many instructions per cycle (IPC) a CPU can execute. However, the CPU's clock cycle and the number of cores are only one factor for a CPUs performance. A key bottleneck that restricts processor speeds is memory. When the processor runs an instruction that requires a memory access into a lower-level cache or main memory due to a miss in the first-level instruction cache, the latency of the access lowers the processing speed. To avoid these misses and reduce latency, hardware methods such as cache replacement policies and instruction prefetching have been designed to achieve a higher IPC resulting in a speedup while using the same physical hardware. Cache replacement policies attempt to keep the most useful data in the cache so that the processor does not need to stall while waiting on an access to main memory. The instruction cache is the first place the processor looks when it needs the next instruction, so having the correct instructions already in the cache produces a speedup. Instruction prefetching attempts to avoid latency from main memory access times by predicting future instructions correctly and fetching them at the right time.

The purpose of this research is to combine instruction prefetchers and cache replacement policies to produce a higher speedup. By surveying a collection of instruction prefetchers and last-level cache replacement policies on a trace-based simulator, speedups for each prefetcher and policy were determined. After determining initial speedups, cache replacement policies were modified to be used on the instruction cache instead of the last-level cache, creating a combination instruction prefetcher and cache replacement policy to improve the processor speed-up. Additionally, we explore utilizing prefetch metadata in the cache replacement policy to improve performance. In this paper, we will discuss the speedup effects of combining certain instruction prefetchers and cache replacement policies in the L1I cache.

ACKNOWLEDGEMENTS

Contributors

We would like to thank our faculty advisor, Dr. Paul V. Gratz, for his guidance throughout the course of this research. Additionally, we would like to thank the Texas A&M Computer Engineering and Systems Group for allowing us to use the CESG cluster for our simulations.

We would like to thank the developers of the ChampSim program as it allowed for us to build upon it to simulate our tests.

Finally, we would like to thank our ECEN 403 and 404 instructors and teaching assistants for helping us work through our project and helping us keep on track.

The simulation traces and code for multiple cache replacement policies and instruction prefetchers were provided by two different competitions: The 2nd Cache Replacement Championship and The 1st Instruction Prefetching Championship, respectively. We want to thank those who developed these policies.

Funding Sources

No funding was needed for this research.

NOMENCLATURE

BRRIP	Bimodal Re-reference Interval Prediction
CESG	Computer Engineering & Systems Group
CFG	Control Flow Graph
ChampSim	ChampSim is a trace-based simulator for a microarchitecture study.
CPU	Central Processing Unit
DRRIP	Dynamic Re-reference Interval Prediction
GB	Gigabytes
IPC	Instructions per Cycle
KB	Kilobytes
KPC	Kill the Program Counter [9]
LIME	Less is More
LRU	Least Recently Used
RAS	Return Address Stack, used to predict a return address from a function call
RDIP	RAS Directed Instruction Prefetching, generates signature from current RAS and associates cache miss with that signature, and prefetches caches according to signatures
RRIP	Re-reference Interval Prediction
RRPV	Re-Reference Prediction Value
SHCT	Signature History Counter Table
SHiP	Signature-based Hit Predictor
SRRIP	Static Re-reference Interval Prediction

Stream Prefetcher	uses a stream buffer that captures addresses of cache misses and a certain number of subsequent addresses so it can prefetch subsequent addresses of an already missed instruction line. Ex: Block A misses, stream buffer fills with A+1, A+2, A+3, ..., A+K by prefetching, then delivers those when requested by L1 Cache
Transitive Closure	the set of nodes that can be traversed to from a starting node, or can be the shortest path between two nodes

1. INTRODUCTION

1.1 Purpose

Over time, processor speeds have exponentially increased while staying efficient in power consumption. However, processing speeds have become bottlenecked by the speed of memory. Accesses into main memory take a long time because main memory is a large data storage. The electrical characteristics of large data storages result in higher read times, so finding the data the instruction calls for in main memory takes a lot longer than it would in a smaller data storage. Main memory is also far away from the processors, causing longer latency while sending messages to and from the processor and main memory. Because of this latency, modern processors use a hierarchy of smaller in size memory storages called caches to reduce memory access latency [4]. The caches located closest to the processors hold the least amount of data so that they can have the fastest access times. Developers of CPUs use this structure to help decrease latency and help bring out the best in their high-speed processing cores. By holding data that will be needed in caches, the speed of processing an instruction can be magnitudes faster than looking for the data in main memory. There are two areas that our research will focus on regard to caches: replacement policies and instruction prefetching.

By researching how performance in processing speeds is affected by cache replacement policies in conjunction with instruction prefetching policies in the L1I cache, a field not yet explored thoroughly, we hope to discover relationships between the policies that help improve the accuracy of cache hits to improve processing speeds.

1.2 Cache Replacement Policies

Cache replacement policies are used to determine what data is held within a cache. Because the space available in caches is more limited compared to main memory to ensure low access times, keeping the data that will be used frequently inside the cache will increase the hit percent on memory searches and is very important and a main area of research to increase processing speeds. Cache replacement policies are tasked with the question of: “When a new line is to be inserted into the cache, which line should be evicted to make space for the new line?” [6]. Thus, replacement policy determines the contents of the cache.

Replacement policies are not the only way of improving the cache hit rate, the rate at which an access into memory successfully finds the data needed in the cache, there are two other methods of increasing this rate. One of the other methods of increasing hit rate is increasing the cache’s size. By having more data in the cache, we effectively increase the likelihood that the data needed is found in the cache. The problem with this method is the increase in access time as the computer must search through more data to find if the data needed is found in the cache. The other method to increase hit rate is increasing the cache’s associativity however this method increases power consumption. Because of the downsides the other methods have, we look at replacement policies to make caches a lot more efficient without changing their size or associativity. [6] The question then becomes: what is the most efficient replacement policy? There are many areas in which replacement policies have area for improvement, such as using branch target buffers to make policies predictive among other areas [1], but in this paper we will look at how pairing existing policies with instruction prefetchers can improve processing speeds.

1.3 Instruction Prefetchers

Instruction misses occur when the processor seeks a target instruction and the instruction cache does not have the instruction already loaded. These misses stall the processor pipeline and decrease processor performance by reducing the number of instructions per cycle [4]. Instruction prefetching is a way to reduce the number of cache misses in a processor's instruction caches. Hardware within the processor can predict which instructions will need to be read by the processor before it tries to access them, which reduces the overall latency of instruction cache miss accesses. A working prefetching mechanism must correctly predict the address of a memory access, correctly predict when to issue a prefetch, and correctly place the prefetched data in the cache [3]. An instruction prefetcher needs to correctly find the right instruction at the right time so it can be executed without any additional miss latency. Many modern workloads involve complex programs with large instruction sets, making instruction prefetching vital in reducing the overall number of instruction stalls in a processor [3].

1.4 Combining Instruction Prefetchers and Cache Replacement Policies

Instruction prefetchers and cache replacement policies are both ultimately speculating about future memory usage and are each limited by their respective algorithms and hardware to try to guess which memory will be used in the future. Typically, instruction prefetchers do not communicate with cache replacement policies, and cache replacement policies do not interact with instruction prefetchers. Prior work has explored holistically combining data prefetching with cache replacement into a system that is greater than the sum of its parts [9], but in this research we explore the idea of using existing instruction prefetch metadata from the instruction prefetcher within an L1I cache replacement policy to improve the combined system's performance more than having them work independently.

2. METHODS

2.1 Tools and Resources

2.1.1 ChampSim

ChampSim, a trace-based computer microarchitecture simulator, was used to test and measure speedup on a single core processor using varying parameters, architectures, and traces. ChampSim runs instruction sets (traces) on a simulated processor with the option to choose a branch predictor, L1I prefetcher, L1D prefetcher, L2C prefetcher, LLC prefetcher, LLC replacement policy, and the number of CPU cores. The simulation process involves compiling a binary from input parameters, simulating the binary with a chosen trace and number of instructions, and checking the output file for results after the simulation concludes. The version of ChampSim that is being used is that of late 2020 and early 2021.

2.1.2 CESG Cluster

Simulations were run using the Computer Engineering & Systems Group cluster. Specifically, the four high performance compute nodes were heavily used for large simulations.

2.1.3 Python

Python3 was used to automate parts of the simulation process for ease. Python scripts were used to run simulations on different cores and to scrape key metrics like Instructions Per Cycle from the result files.

2.1.4 The 1st Instruction Prefetching Championship

Instruction Prefetcher models from The 1st Instruction Prefetching Championship were used in this research. The purpose of this competition was to compare different instruction prefetching algorithms that had a fixed storage budget of 128 KB.

The prefetchers from this competition were simulated to observe processor speedup, modified to fit under a storage size of 32 KB and simulated again, and then combined with cache replacement policies for the purpose of producing a higher speedup.

2.1.5 The 2nd Cache Replacement Championship

Cache replacement policies from the Texas A&M hosted replacement championship were used in this research. The purpose of this competition was to create cache replacement policies to improve IPC speeds is the L2 cache. The policies were built to work on an old and modified version of ChampSim.

The policies from this competition were modified to work with the newest version of ChampSim as well as to work on the L1I cache.

2.2 Individual Simulations

2.2.1 Cache Replacement Policies

The cache replacement policies used in this research were modified to work with the newest version of ChampSim, which involved modifying the functions and algorithms that were meant to work with a specific version of ChampSim while keeping the policies working as intended. These modifications needed to be made to ensure that the policies would be able to work in conjunction with the instruction prefetching policies also used in this research.

Additionally, all the replacement policies and prefetchers need to work on the L1I cache for the scope of this research, so all the replacement policies needed to be modified to be able to work in the L1I cache.

All the cache replacement policies will use LRU as a baseline for measuring speedup as it is one of the most basic replacement policies used in today’s computing environment.

2.2.1.1 Bélády’s Min

The Bélády’s Min algorithm is meant to be the optimal cache replacement policy that reduces the amount of cache miss rate to the lowest value possible. The algorithm is built on the idea that the cache holds information that will be used soon and evicts data that will not be used in the near future. Bélády’s Min will provide some insight as to improvements that could be done to LRU to improve the combination policies in the L1I cache. Bélády’s algorithm is used as a foundation for many other cache replacement policies as it is theoretically the ideal policy.

2.2.1.2 LRU [7]

LRU stands for Least Recently Used. LRU is usually a baseline in terms of replacement policies because of its relatively simple algorithm. LRU decides what data is held within the cache by looking at what data has been least recently used and replaces it with data that has been used by the current instruction. The idea behind this policy is that if data was used recently it will most likely be used again, so housing it in the cache will speed up the next time it is needed.

2.2.1.3 SHiP [15]

SHiP stands for Signature-based Hit Predictor. SHiP uses a signature table to predict what data will receive cache hits in the future. By using a signature table, SHiP can use a counter method to keep track of the frequency use of all the cache’s data so that it can evict and add data that will be more likely to be used soon, to avoid cache misses.

2.2.1.4 SRRIP [7]

SRRIP stands for Static Re-reference Interval Prediction and is based off the policy named RRIP that uses a linked list of sorts to hold data that will most likely be referenced soon. SRRIP also holds a counter that informs the updates of the cache, this counter is called the RRPV (Re-Reference Prediction Values). The head of the list in the cache that is believed to be referenced soon holds a 0 in the RRPV while the tail consists of data that is believed to be used long in the future, which has an RRPV equal to the max RRPV. The linked list allows for the near-future predicted data, the head of the list, to be accessed as quickly as possible. When a hit occurs, the data that was accesses has its RRPV set to 0 as it is expected to be re-referenced soon. When a miss occurs, all RRPVs are incremented and the first 3 in the list, starting from the head, is replaced by the new data and set with an RRPV of the max RRPV minus 1. This method avoids the pitfall of evicting the newest data only because it is at the tail of the list.

2.2.1.5 DRRIP [7]

DRRIP stands for Dynamic Re-reference Interval Prediction. DRRIP is based on RRIP similar to SRRIP as it dynamically chooses between two RRIP based replacement policies to avoid issues like scanning, bursts of references to memory that reference memory that was predicted to be referenced long into the future, creating long access times within the cache. and thrashing, the overuse of a computer's virtual memory, that are often pitfalls for replacement policies. DRRIP chooses between SRRIP and BRRIP (Bimodal Static Re-reference Interval Prediction) to avoid these pitfalls when possible. Based on the workload the processor is undergoing DRRIP decides what policy would yield the best cache hit rate.

2.2.1.6 SHiP++ [16]

SHiP++ is a proposed enhancement to SHiP that won at the 2nd Cache Replacement Championship, with the highest speed-up of all entered policies. The policy proposed five enhancements to the original SHiP. The first enhancement involves inserting data into the cache that is typically referenced along with data that was just referenced by a recent instruction. This data is also maintained with a counter to ensure that it is not evicted immediately after not being used. The second enhancement was made to the SHCT, the table that holds the memory addresses for the data in the cache is held, to weigh cache hits and evictions similarly to avoid cache misses in the future. The third enhancement is improving the writeback awareness of the policy. Because writebacks are often not re-referenced, evicting them sooner rather than latter will help with unused memory in the cache. The fourth enhancement is adding special signatures to prefetched data so that the policy can make a distinction and learn the behavior of prefetched data. The fifth enhancement again deals with prefetched data. The policies treat the counters on the prefetched data differently to ensure that the policy can learn how to use the data.

2.2.1.7 Less Is More (LIME) [14]

LIME adopts Bélády Trainer's algorithm but does not use certain aspects of the algorithm that were deemed unnecessary. LIME uses the history of cache hits and accesses to assess what will be inserted into the cache. LIME samples 20 random sets of instructions and filling the cache based on the findings. The Bélády algorithm focuses on looking into history to see what data is typically re-referenced and chooses to occupy the cache with that data and evict the rest. LIME keeps the load and store algorithm aspects of the Bélády algorithm while abandoning many of the other features.

2.2.2 Instruction Prefetchers

2.2.2.1 Preliminary Simulations

All competition prefetchers listed below were simulated under their competition parameters on a single core processor with a bimodal branch predictor and no other prefetchers or cache replacement policies. Simulations were done using 50 unique traces that included server and client workloads with a 50 million instruction warmup to populate the tables in the prefetcher hardware followed by 50 million test instructions.

Next, all prefetchers were modified to fit under a budget of 32 KB to match more realistic microarchitecture storage space.

2.2.2.2 EIP (Entangling Instruction Prefetcher) [12]

The Entangling Instruction Prefetcher uses tables to track the time it takes between a demand access instruction call and when the block arrives in the L1I cache. Next, it uses this timestamp to check its instruction table to find which source instruction needed the missed destination instruction. It then entangles the source and destination instructions and records an entry in the entangled table so that in the future it can prefetch the destination instruction each time its source instruction is executed or prefetched.

To fit within a 32 KB size, the Entangled Table which tracks which instructions are entangled was reduced from 113 KB to 14 KB. EIP was then simulated again with the same conditions as before.

2.2.2.3 FNL+MMA (Footprint Next Line and Multiple Miss Ahead Prefetcher) [13]

The FNL+MMA prefetcher combines two prefetcher concepts with complimentary prefetching tradeoffs. The Footprint Next Line prefetcher prefetches sequential next instruction lines by associating cache blocks to each other using a Touched Table and a WorthPF Table. The Touched Table tracks whether a cache block has been touched recently by a demand access, while the WorthPF table is a 2-bit entry that tracks which lines are worth prefetching. These tables track cache blocks that trigger demand accesses of up to the next 5 cache blocks. The multiple miss ahead prefetcher overcomes the sequential limitation of the FNL prefetcher using a special cache called the Instruction Shadow cache, which is a tag-only table that tracks only demand accesses. When the MMA prefetcher sees that both a cache block N is typically the Nth miss after block 1 AND block N misses the instruction cache, it associates them in the Miss Prediction Table. The

MMA then prefetches block N any time it calls block 1 because it has associated the two blocks. The prefetcher in the competition used FNL5+MMA9 which means it prefetches up to 5 next lines and has an MMA with an ahead distance of 9 blocks.

To reduce the FNL+MMA size to 32 KB, three tables were reduced to a quarter of their original size. The Touched and WorthPF Tables were reduced from 8 KB and 16 KB to 2 KB and 4 KB, respectively. The Miss Ahead Prediction Table, which tracks associated addresses to be prefetched was reduced from 71 KB to 18 KB.

2.2.2.4 D_JOLT (Distant Jolt Prefetcher) [11]

The Distant Jolt Prefetcher combines a long-range prefetcher that predicts far away instructions with high coverage, a short-range prefetcher, and a fallback prefetcher that predicts instructions in the near future with high accuracy. The long and short-range prefetchers use variations of RDIP. They use a RAS to predict return addresses from function calls. The fallback prefetcher is a stream prefetcher that will prefetch later after the short and long-range prefetchers fail to predict addresses and cause cache misses.

2.2.2.5 Barça (Branch Agnostic Region Searching Algorithm) [8]

The Branch Agnostic Region Searching Algorithm treats cache blocks and block groups as nodes on a graph. By tracking block traversal during control flow as edge weights, the algorithm can later find candidates to prefetch using probabilities calculated by the product of edge weights and a depth-limited DFS. This algorithm is branch agnostic because it determines prefetches from the control flow demand fetches to different block regions.

To reduce Barça's storage overhead, the CFG data structure was reduced from 104KB to 26KB, bringing the total size to 32.48KB.

2.2.2.6 PIPS (Prefetching Instructions with Probabilistic Scouts) [10]

The Prefetching Instructions with Probabilistic Scouts algorithm uses the concept of a control flow graph and treats each memory line as a node and each probability of traversal from one node to the next as an edge. Next, “scouts” are sent out to explore the graph by traversing according to probabilities stored in a table called the Line History Table. The scouts prefetch memory lines when they encounter their corresponding nodes during path traversal. After a certain number of steps, the scouts die, and new scouts are sent out from the front of the line which is the current instruction.

2.2.2.7 MANA (Microarchitecting an Instruction Prefetcher) [2]

The Microarchitecting an Instruction Prefetcher algorithm tries to illustrate how choosing metadata carefully and microarchitecting the metadata storage can result in a small prefetcher with a high speedup. This prefetcher is different from other modern prefetchers such as RDIP, Shotgun, and PIF because it avoids the large storage overhead those prefetchers all have. The algorithm creates spatial regions for cache lines and stores them as entries in a set associative table. Each region points to its successor in another table entry, and MANA issues prefetches for each region’s successors.

2.2.2.8 TAP (Temporal Ancestry Prefetcher) [5]

The Temporal Ancestry Prefetcher uses a control flow graph by approximating cache lines as nodes and finding transitive closures between those nodes. The TAP algorithm runs alongside next line prefetchers. Temporal prefetching tries to predict future cache misses using an ancestry table that tracks old cache misses and enters them as descendants of instruction addresses. On any cache access, the TAP algorithm prefetches the descendants of the current instruction in the program counter. The ancestry table uses weight values to determine how far into the ancestry table to go to find descendants. These weight values are scaled with algorithm performance, meaning weights are tempered when cache blocks are evicted without being executed or incremented when they are useful.

2.3 Combined Simulations

2.3.1 Prefetchers

The prefetchers were ported to work with a development branch of ChampSim that supports changing cache replacement policies in the L1I cache. Unfortunately, the Entangling Instruction Prefetcher was not able to be ported over and is left out of future simulations. Additionally, the development branch of ChampSim uses LRU in the L1I cache by default, which means that the new baseline for speedup is no prefetcher with an LRU cache replacement policy in the L1I cache.

2.3.2 Cache Replacement Policies

ChampSim does not originally support a change in the working cache replacement policy for any caches other than the last level cache (LLC). By modifying the code of the developmental branch of ChampSim we were able to introduce different cache replacement policies apart from the default LRU. Moving the replacement policies to the L1I cache meant that the policies would be working on a smaller sized cache than the typical lower level data caches that the policies typically run in.

2.3.3 Simulations

Simulations were run by varying the instruction prefetcher and cache replacement policies on the L1I cache, using the same traces as in prior simulations and keeping other parameters constant. Simulations will consist of the top three instruction prefetchers, FNL-MMA, Barça, and

PIPS, after their size reduction and all the tested cache replacement policies, LRU, SRRIP, DRRIP, SHiP, SHiP++, and LIME.

2.4 Modifying Combination Policy

2.4.1 Purpose

Without modifications to the prefetchers or cache replacement policies, the results of the combination policies showed no improvement over the LRU combination baselines. We attempted to reduce interference between the prefetchers and cache replacement policies by having them communicate. Communication between data prefetchers and the last-level cache has been explored in KPC [9], but as far as we are aware communication of prefetch metadata between an instruction prefetcher and L1I cache replacement policy has not been researched. We aimed to choose a prefetcher and cache replacement policy combination that should have been promising according to individual simulation results and contained pre-existing functionality that could be exploited with little additional overhead for an improved speedup. The Barça prefetcher was the second best prefetcher in individual testing, and SRRIP was the best tested cache replacement policy according to simulations in the L1I. Since Barça generates probability values for each of its prefetches and SRRIP utilizes a modifiable RRPV value in its algorithm, this combination was also conducive to adding communication between the prefetcher and cache replacement policy.

2.4.2 Barça-SRRIP Combination

As part of its prefetching algorithm, Barça traverses and prefetches nodes on a weighted control flow graph and assigns probabilities to each prefetch. As part of its algorithm, Barça tracks traversal between block regions as graph edges for which it assigns weights equivalent to the number of times it was traversed. The weights of edges originating at a given node are used to determine a probability that one of those edges will be traversed, meaning Barça has readily available probability metadata for each of its prefetches. As Barça issues depth first searches into the graph, it adds blocks from each region it encounters into a list of prefetch candidates, only deciding to prefetch candidates that meet a certain probability threshold. SRRIP utilizes an RRPV for each cache line to promote cache lines on hits (decreasing RRPV) and demote them on misses (increasing RRPV), evicting cache lines with the highest RRPV value when a new line must be placed.

Our implementation attempts to utilize Barça's prefetch probabilities within a modified SRRIP algorithm to assign low confidence prefetches with higher RRPVs and higher confidence prefetches with lower RRPVs. We wanted to allow higher probability prefetches to stay in the cache slightly longer to reduce the likelihood that they are removed from the cache before they are used. We used an iterative approach to find an algorithm that was most helpful in providing a higher speedup.

Figure 2.4.2.1 depicts the use of confidence values to assign RRPVs to each prefetch, and how the proposed algorithm would handle evictions. Using the algorithm, data0 was able to stay within the cache despite being the least recently used prefetch when compared to the other prefetches. By assigning RRPVs based on the confidence value, the cache is more likely to hold prefetches who are more likely to be used in the cache longer than those who have a less likely chance of being used.

Incoming Prefetch	Confidence Value	Hit/Miss	Way0	Way1	Way2	Way3
data0	90%	Miss	Data: RRPV: 3	Data: RRPV: 3	Data: RRPV: 3	Data: RRPV: 3
data1	30%	Miss	Data: data0 RRPV: 0	Data: RRPV: 3	Data: RRPV: 3	Data: RRPV: 3
data2	67%	Miss	Data: data0 RRPV: 0	Data: data1 RRPV: 2	Data: RRPV: 3	Data: RRPV: 3
data3	45%	Miss	Data: data0 RRPV: 0	Data: data1 RRPV: 2	Data: data2 RRPV: 1	Data: RRPV: 3
data2	67%	Hit	Data: data0 RRPV: 0	Data: data1 RRPV: 2	Data: data2 RRPV: 1	Data:data3 RRPV: 2
data4	10%	Miss	Data: data0 RRPV: 0	Data: data1 RRPV: 2	Data: data2 RRPV: 0	Data:data3 RRPV: 2
data0	90%	Hit	Data: data0 RRPV: 1	Data: data4 RRPV: 2	Data: data2 RRPV: 1	Data:data3 RRPV: 3
data5	80%	Miss	Data: data0 RRPV: 0	Data: data4 RRPV: 2	Data: data2 RRPV: 1	Data:data3 RRPV: 3

Figure 2.4.2.1 Use of Confidence Values to Assign RRPVs

3. RESULTS

3.1 Individual Simulations

3.1.1 Cache Replacement Policies

Figure 3.1.1. shows the speedups of the tested cache replacement policies on the L1I cache. Based on the results we see that the current highest speedup is held by SRRIP and DRRIP. Additionally, both SHiP and SHiP++ lost performance in terms of speedup over LRU on the L1I cache. The complexity of some of the replacement policies lend themselves more towards working on the LLC, a larger sized cache. Having to work with such a small cache, SHiP and SHiP++ are unlikely to fully utilize their signature-based hit prediction system making it so that they do not evict the proper instructions when the cache needs to add something into the cache, decreasing the overall usefulness of the L1I cache creating lower IPCs.

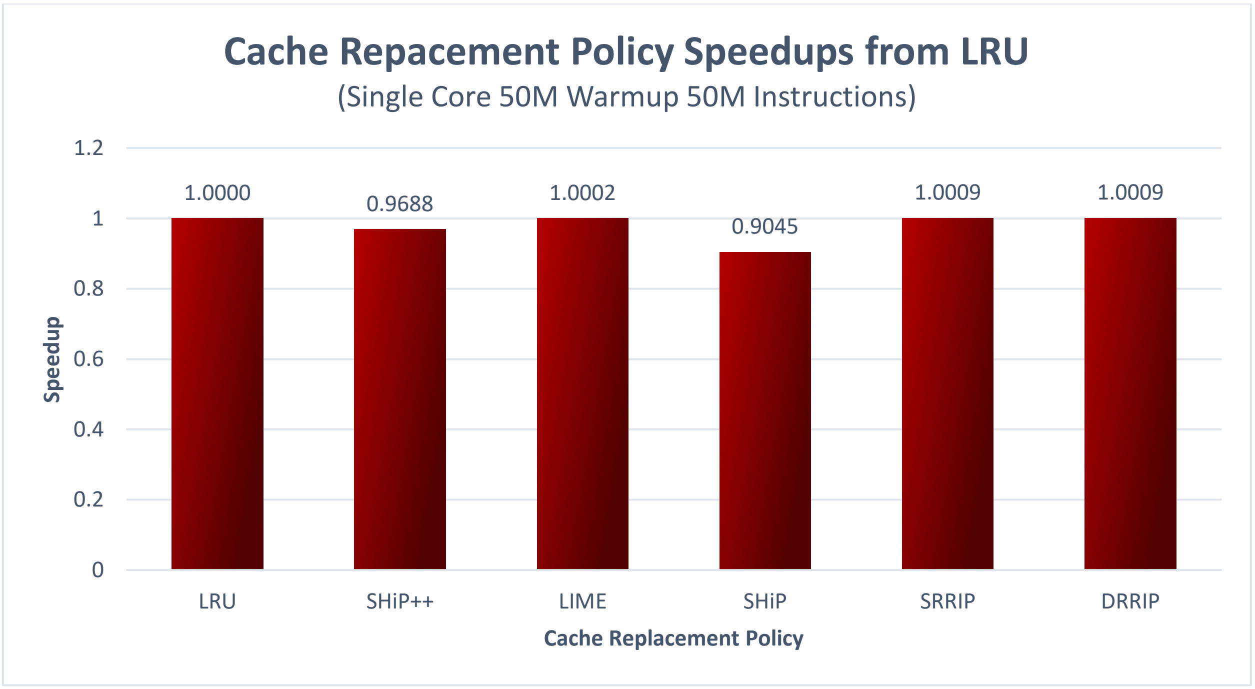

Figure 3.1.1. IPCs and Speedups for all Cache Replacement Policies tested on the L1I Cache

3.1.2 Instruction Prefetchers

The simulation of instruction prefetchers from the IPC1 in Figure 3.1.2. show that the top performing prefetchers were EIP, FNL+MMA, and D_JOLT.

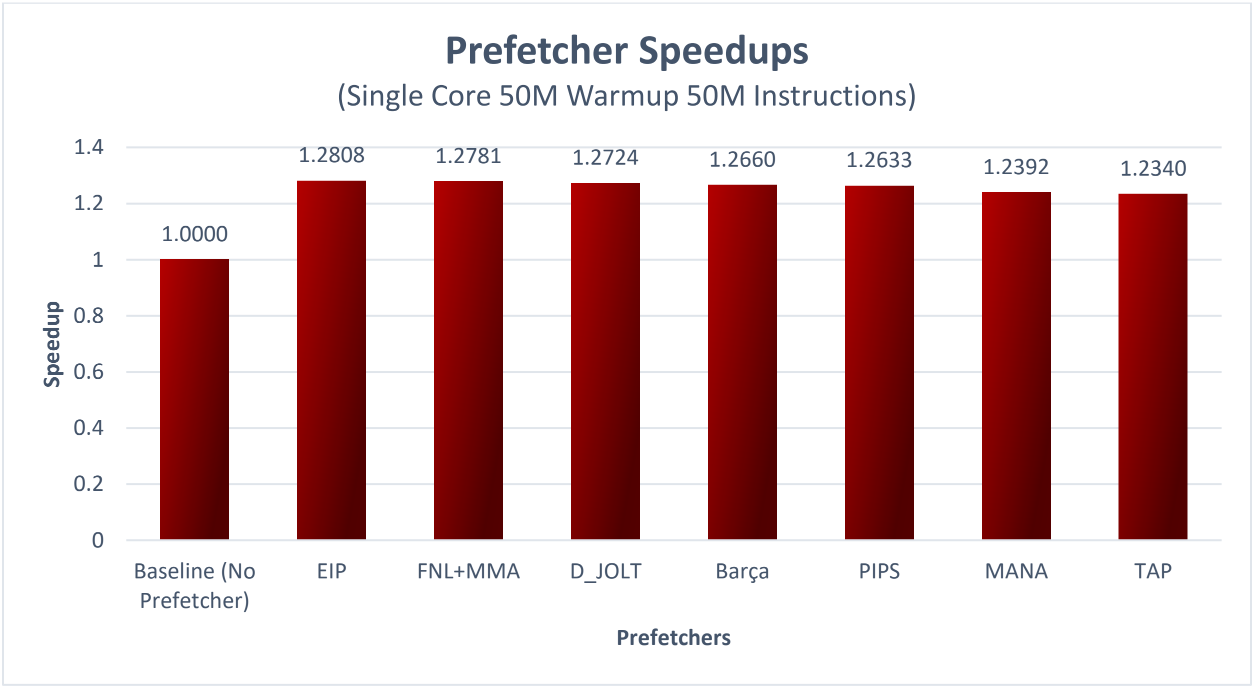

Figure 3.1.2. Speedups across Competition Prefetchers on 50 Traces

These simulation results agree with the original IPC-1 results which stated that the best speedups were achieved by EIP, FNL+MMA, and D_JOLT, in order.

3.1.3 Instruction Prefetchers: Reduced Size

To create a more realistic system for testing the combination of instruction prefetchers and cache replacement policies, these large prefetchers were reduced in size to a more reasonable 32KB soft limit. In Figure 3.1.4., the results of the simulations for the reduced prefetchers showed that FNL+MMA, Barça, and PIPS are the best performing reduced prefetchers.

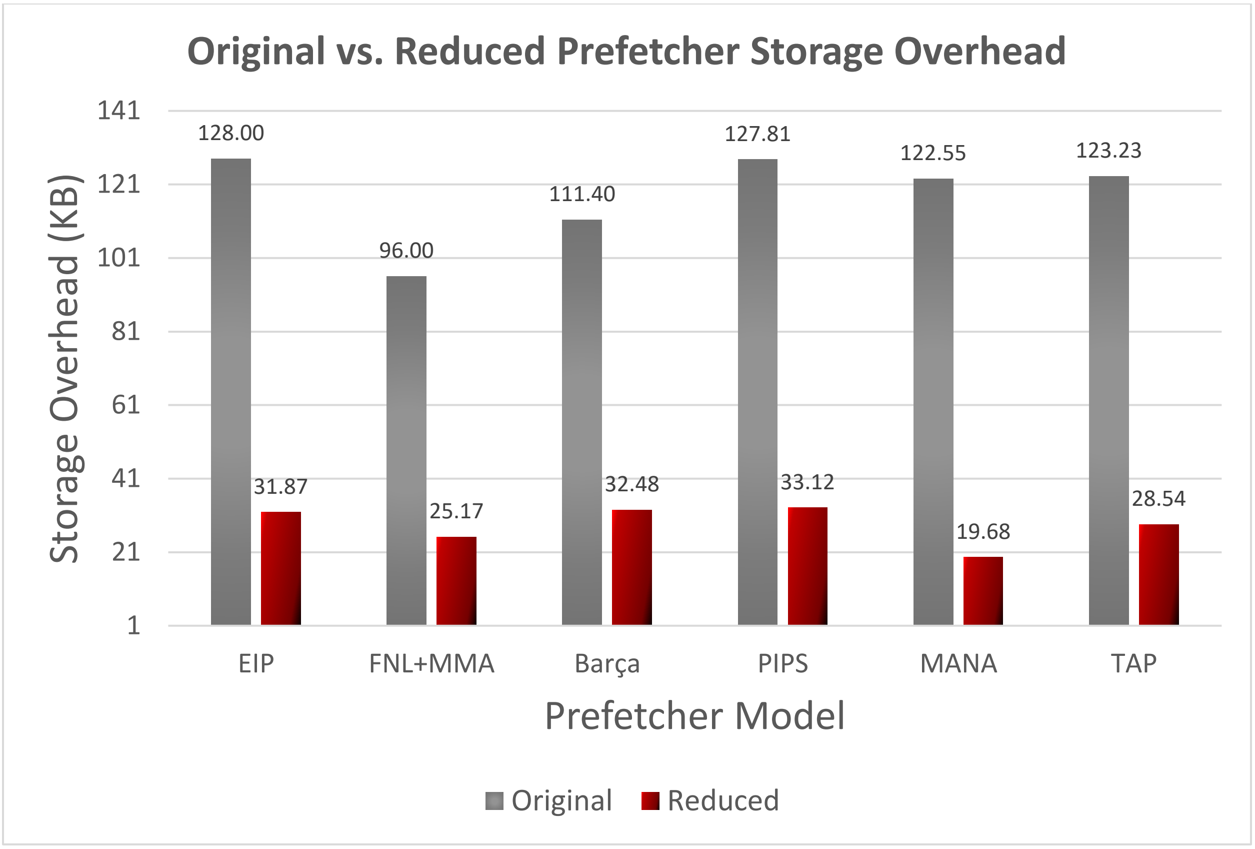

Figure 3.1.3. Storage Overhead for Unmodified and Reduced Prefetchers

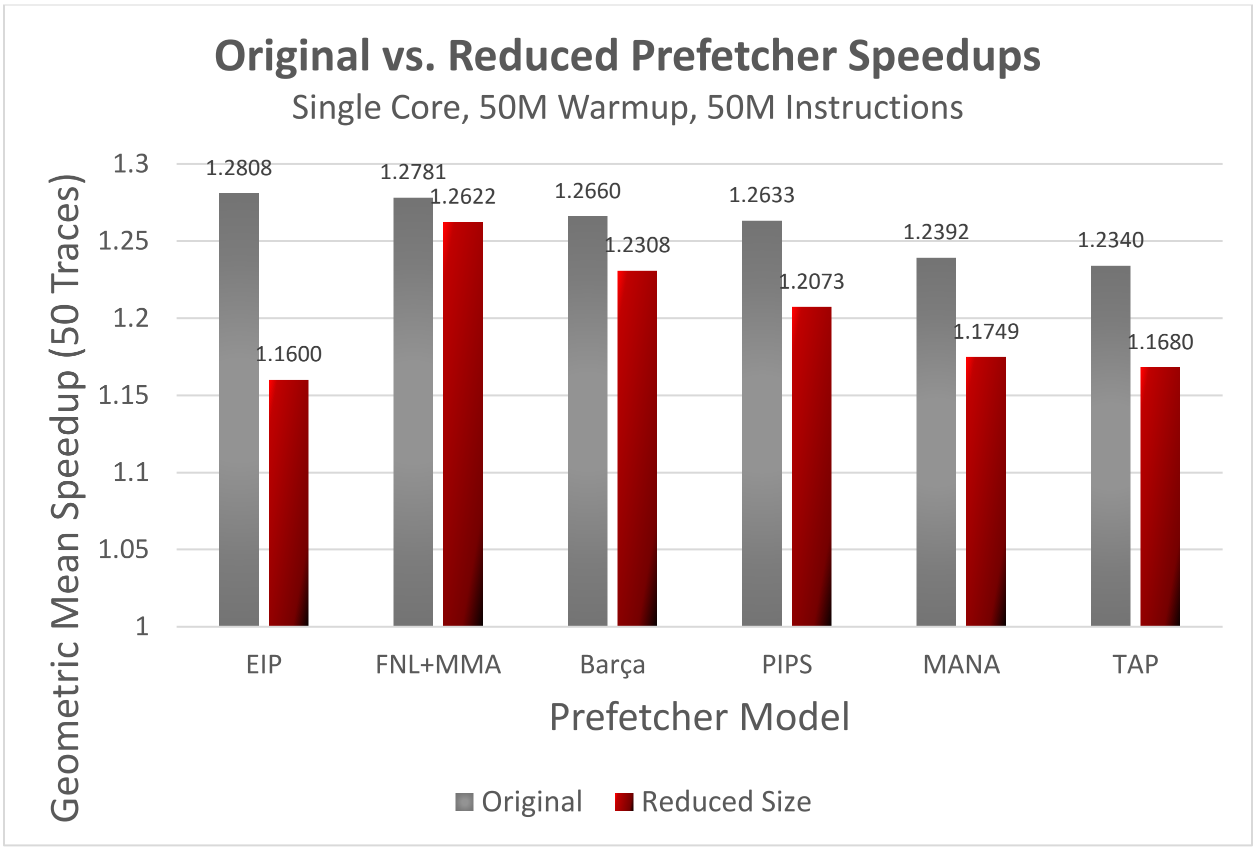

Figure 3.1.4. Unmodified and Reduced Prefetcher Speedups

Some of the reduced prefetchers took a major hit to performance, while others scaled well. EIP, previously the best performing prefetcher, became the worst performing prefetcher. EIP's reduced performance is not surprising because it was built with large overhead in mind to track many entangled instruction addresses. D_JOLT was not reduced due to unresolvable errors.

3.2 Combined Simulations

The results of combining the cache replacement policies with the top three performing instruction prefetchers are graphed below in Figure 3.2.1. In each prefetcher case, the combination with LRU provides a better speedup than any other cache replacement policy, indicating that these cache replacement policies are not well suited for the L1I cache. Additionally, the overall speedup values for the combinations appear lower than the individual simulation results in Figure 3.1.4.

because the individual simulations were compared to a baseline with no L1I cache replacement policy whereas the baseline in Figure 3.2.1. uses LRU for the L1I cache replacement policy. Since the combination simulations were normalized against a system with no prefetcher and LRU in the L1I cache, their speedups are lower than they were in Figure 3.1.4. because those values were normalized against a system with no prefetcher and no cache replacement policy in the L1I cache.

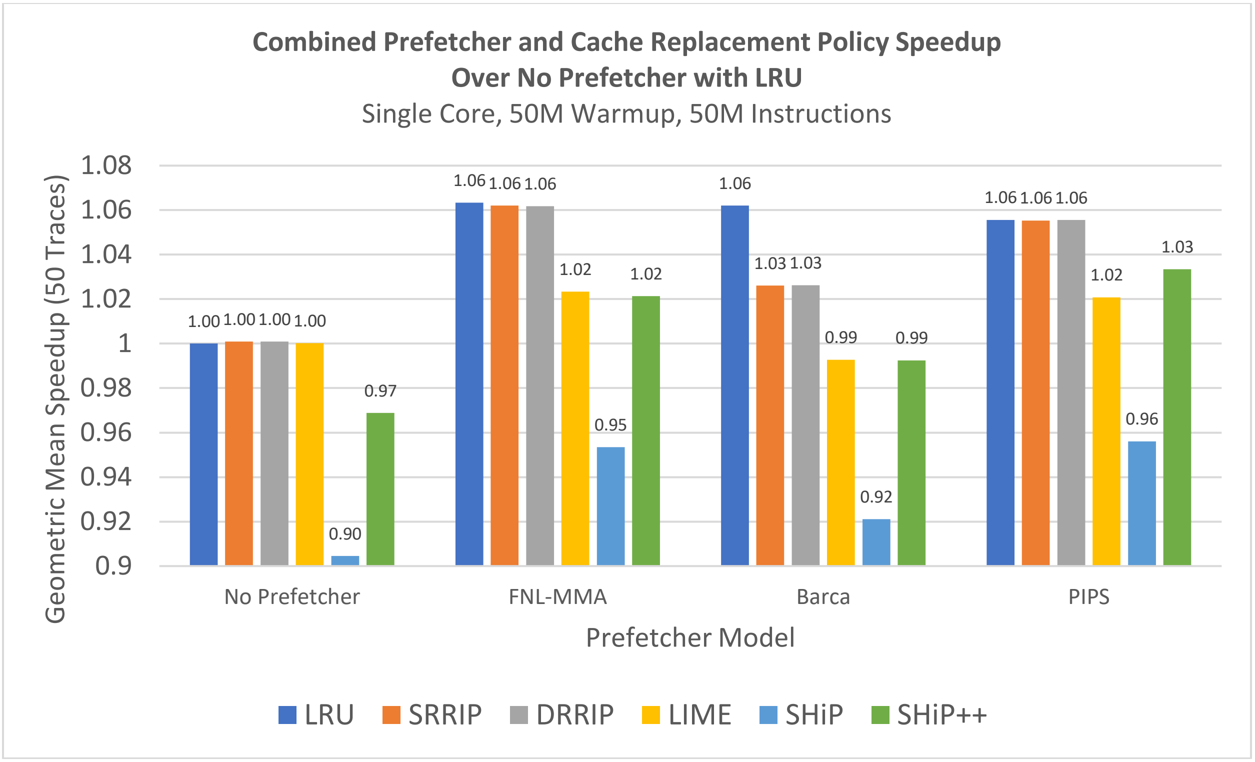

Figure 3.2.1. Combined Prefetcher and Cache Replacement Policy Speedups

3.3 Modified Barça SRRIP Results

To test the modified Barça-SRRIP combination policy we made iterative tests for both of our ideas of the introduction of confidence values to the cache replacement policy and updating hit prefetches' RRPVs to push them towards eviction. In Figure 3.3.1 the speedups of the modified combination policy can be seen when we introduce confidence values for every prefetch to the cache replacement policy. It also shows the speedups of the combination policy with a modification to alter the RRPVs of prefetches by demoting the cache line if a prefetch causes a cache hit. The speedup of the modified combination showed a small improvement over the combination with no communication, and the IPC was lowered in the model with prefetch hit demotion.

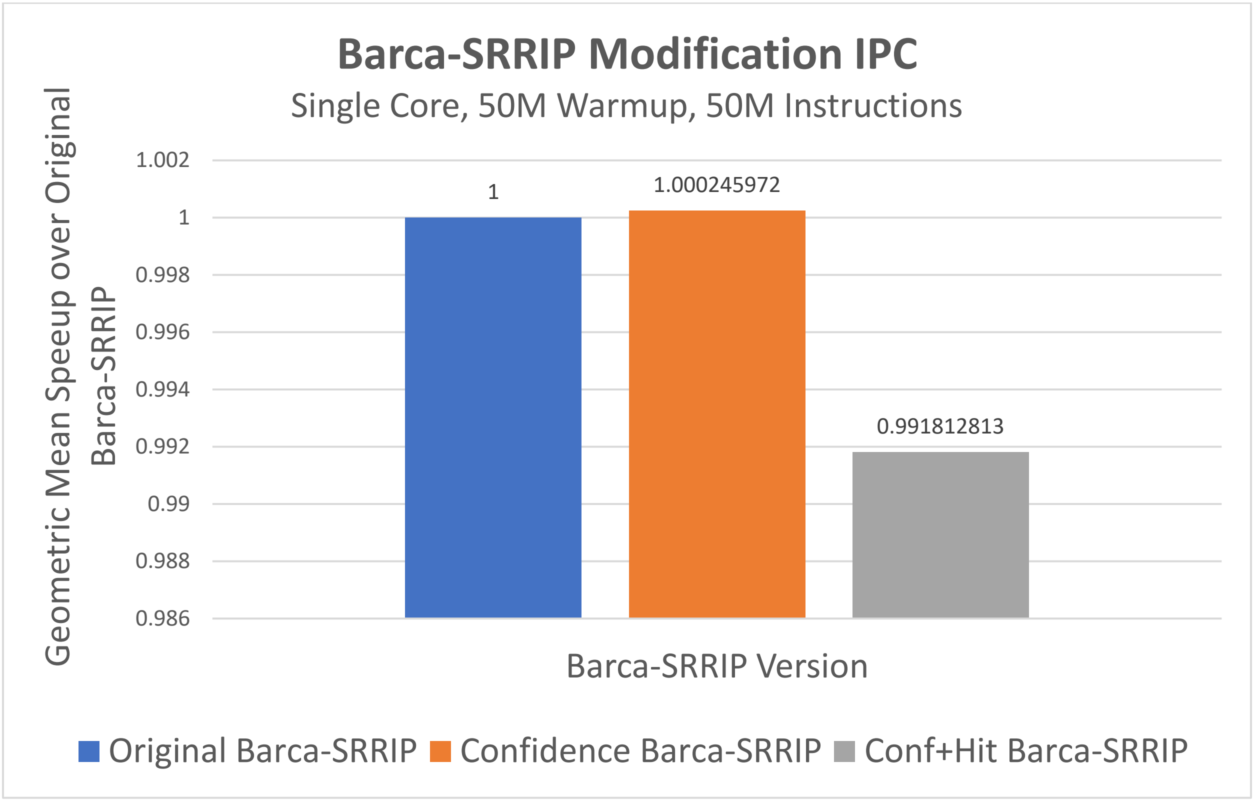

Figure 3.3.1. Results from the Barca-SRRIP Modifications

3.4 Analysis of Cache Replacement Policies combined with Instruction Prefetchers

The initial simulations of individual prefetchers and cache replacement policies provided performance metrics used to find their speedups. After reducing the instruction prefetchers and simulating them again, the best prefetchers were, in order, FNL+MMA, Barça, and PIPS.

The results of the combination simulations between the top three prefetchers and the cache replacement policies show a few things. First, the prefetchers maintained their order of performance relative to each other. Second, the highest speedups for each prefetcher were achieved with LRU, not with any of the other cache replacement policies being tested. Lastly, despite providing a slight speedup over LRU in the case of no prefetcher, SRRIP and DRRIP performed worse than LRU when combined with instruction prefetchers, indicating that the cache replacement is interfering with the prefetched instructions.

The results also show that naively combining instruction prefetchers and cache replacement policies in the L1I cache does not yield significant improvement and can even lead to a reduction in performance. Because the L1I cache is so small, both the instruction prefetcher and the cache replacement policy have to mesh well together to ensure that the cache is holding the proper instructions and evicting the instructions it no longer needs. If the two policies do not work well together by evicting the data/instructions that the other policy needs, we end up with a situation like the combination policy of Barça-LIME. Barça-LIME had a decrease in performance when compared to LIME without a prefetcher. This situation shows that not all combination policies will yield an increase in performance when compared to the instruction prefetchers or cache replacement policies on their own.

To improve performance using combination policies we propose the use of modified combination policies that pass along prefetch metadata from the instruction prefetcher to the cache replacement policy. This modification allows for informed updates by the cache replacement policy based on the information gathered by the instruction prefetcher. In the next section, we will analyze the results from the modified Barça-SRRIP combination policy, where we passed prefetch confidence values from Barça to inform the updates made to the cache by SRRIP.

3.5 Analysis of Modifying the Barça-SRRIP Combination Policy

The modified Barça-SRRIP combination policy did not provide a significant speedup over the version without communication between Barça and SRRIP. The final speedup was minimal despite iteratively tweaking the probability thresholds for RRPV assignments. However, because a speedup was achieved, as minimal as it was, a conclusion can be drawn about modified combination policies that send prefetch metadata from the instruction prefetcher to the cache replacement policy. If the algorithm for the cache replacement policy is optimized in a way that best works with the instruction prefetcher and can have informed cache updates based on prefetch metadata, a higher speedup can be achieved.

4. CONCLUSION

Our research explored the idea of combining two common techniques, instruction prefetching and cache replacement policies, to improve processor performance. The initial combination results in Figure 3.2.1. did not show that any specific kind of instruction prefetcher or cache replacement policy naturally worked well together when naively combined. It is likely that the LRU combinations produced the highest speedup simply because the other cache replacement policies, which were designed to work on the last level cache, were hindered by the small size of the L1I cache and could not make full use of their algorithms.

We next worked to try to utilize preexisting instruction prefetch metadata from an instruction prefetcher, Barça, to inform prefetch placement in a cache replacement policy, SRRIP, to improve the performance of the overall system. In Figure 3.3.1., the results show that the introduction of communication between the instruction prefetcher and cache replacement policy did yield a speedup over the system where the two techniques worked independently. It is likely that with algorithmic improvement and fine tuning, this combination system could be improved even more. This research demonstrates a proof of concept that performance can be gained with communication between these performance techniques.

REFERENCES

[1] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer," in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 1-6 June 2018 2018, pp. 519-532, doi: 10.1109/ISCA.2018.00050.

[2] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "MANA: Microarchitecting an Instruction Prefetcher," ArXiv, vol. abs/2102.01764, 2021.

[3] B. Falsafi and T. F. Wenisch, "A Primer on Hardware Prefetching," Synthesis Lectures on Computer Architecture, vol. 9, no. 1, pp. 1-67, 2014/05/31 2014, doi: 10.2200/S00581ED1V01Y201405CAC028.

[4] P. R. Gade, R. Paily, and Y. Ha, "A Branch Target Instruction Prefetching Technique for Improved Performance," in 15th International Conference on Advanced Computing and Communications (ADCOM 2007), 18-21 Dec. 2007 2007, pp. 345-351, doi: 10.1109/ADCOM.2007.101.

[5] N. Gober, G. Chacon, D. Jiménez, and P. V. Gratz, "The Temporal Ancestry Prefetcher," 2020.

[6] A. Jain and C. Lin, "Cache Replacement Policies," Synthesis Lectures on Computer Architecture, vol. 14, no. 1, pp. 1-87, 2019/06/17 2019, doi: 10.2200/S00922ED1V01Y201905CAC047.

[7] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," SIGARCH Comput. Archit. News, vol. 38, no. 3, pp. 60–71, 2010, doi: 10.1145/1816038.1815971.

[8] D. A. Jiménez, P. V. Gratz, G. Chacon, and N. Gober, "BARÇA: Branch Agnostic Region Searching Algorithm," 2020.

[9] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy," SIGARCH Comput. Archit. News, vol. 45, no. 1, pp. 737–749, 2017, doi: 10.1145/3093337.3037701.

[10] P. Michaud, "PIPS: Prefetching Instructions with Probabilistic Scouts," 2020.

[11] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-JOLT: Distant Jolt Prefetcher," 2020.

[12] A. Ros and A. Jimborean, "The Entangling Instruction Prefetcher," IEEE Computer Architecture Letters, vol. 19, pp. 84-87, 2020.

[13] A. Seznec, "The FNL+MMA Instruction Cache Prefetcher," 2020.

[14] J. Wang, L. Zhang, R. Panda, and L. John, "Less is More : Leveraging Belady's Algorithm with Demand-based Learning," 2017.

[15] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer, "SHiP: Signature-based Hit Predictor for high performance caching," 2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 430-441, 2011.

[16] V. Young, C.-C. Chou, A. Jaleel, and M. K. Qureshi, "SHiP + + : Enhancing Signature-Based Hit Predictor for Improved Cache Performance," 2017.