PERCEPTRON LEARNING IN CACHE MANAGEMENT AND PREDICTION TECHNIQUES

A Thesis

by

ESHAN BHATIA

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Co-Chairs of Committee,	Paul V. Gratz
	Daniel A. Jiménez
Committee Members,	Sunil P. Khatri
Head of Department,	Miroslav M. Begovic

May 2019

Major Subject: Computer Engineering

ABSTRACT

Hardware prefetching is an effective technique for hiding cache miss latencies in modern processor designs. An efficient prefetcher should identify complex memory access patterns during program execution. This ability enables the prefetcher to read a block ahead of its demand access, potentially preventing a cache miss. Accurately identifying the right blocks to prefetch is essential to achieving high performance from the prefetcher.

Prefetcher performance can be characterized by two main metrics that are generally at odds with one another: coverage, the fraction of baseline cache misses which the prefetcher brings into the cache; and accuracy, the fraction of prefetches which are ultimately used. An overly aggressive prefetcher may improve coverage at the cost of reduced accuracy. Thus, performance may be harmed by this over-aggressiveness because many resources are wasted, including cache capacity and bandwidth. An ideal prefetcher would have both high coverage and accuracy.

In this thesis, I propose Perceptron-based Prefetch Filtering (PPF) as a way to increase the coverage of the prefetches generated by a baseline prefetcher without negatively impacting accuracy. PPF enables more aggressive tuning of a given baseline prefetcher, leading to increased coverage by filtering out the growing numbers of inaccurate prefetches such an aggressive tuning implies. I also explore a range of features to use to train PPF's perceptron layer to identify inaccurate prefetches. PPF improves performance on a memory-intensive subset of the SPEC CPU 2017 benchmarks by 3.78% for a single-core configuration, and by 11.4% for a 4-core configuration, compared to the baseline prefetcher alone.

DEDICATION

To my mother, my father, and my sister.

ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to my advisors – Prof. Daniel Jiménez and Prof. Paul Gratz. I was really lucky to have two of the best minds in the field of computer sciences working with me. Prof. Jimémez's guidance in the early stages of the project and his strong intuition about the working of caches and perceptrons helped in giving the correct direction to my thesis. Prof. Gratz's experience with prefetching was really instrumental in helping me understand finer nuances about the field. His never ending patience helped me remain motivated even when the results would be far from promising. Prof. Sunil Khatri, also on my thesis committee, strongly motivated me to think about possible constraints in a real hardware implementation of my research.

I can't be more thankful to Gino Chacon, a fellow graduate student at Texas A&M. He made sure he was always available to help me out - be it solving any simulator issues or be it documenting the work. I would also like to thank my collaborators, Dr. Seth Pugsley and Dr. Elvira Teran. Their insighful discussions during the course of the thesis really helped me make this work more novel.

On a more personal note, I would like thank my close friends at Texas A&M – Fazia, Swati, Nalin, and Saumil for being with me through thick and thin of the life as a graduate student. Lastly and most importantly, I am eternally grateful to my family – my parents and my sister, for always being there as pillars of moral support.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Prof. Paul V. Gratz (ECEN Department) and Prof. Daniel A. Jiménez (CSCE Department) as the co-advisors and Prof. Sunil P. Khatri (ECEN Department) as the committee member.

Some contributions in finalizing the overall architecture of the algorithm described in this work were made by the collaborators – Gino Cachon, Dr. Seth Pugsley and Dr. Elvira Teran.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

Graduate study was partly supported by the merit based scholarship from the ECEN Department at Texas A&M University.

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

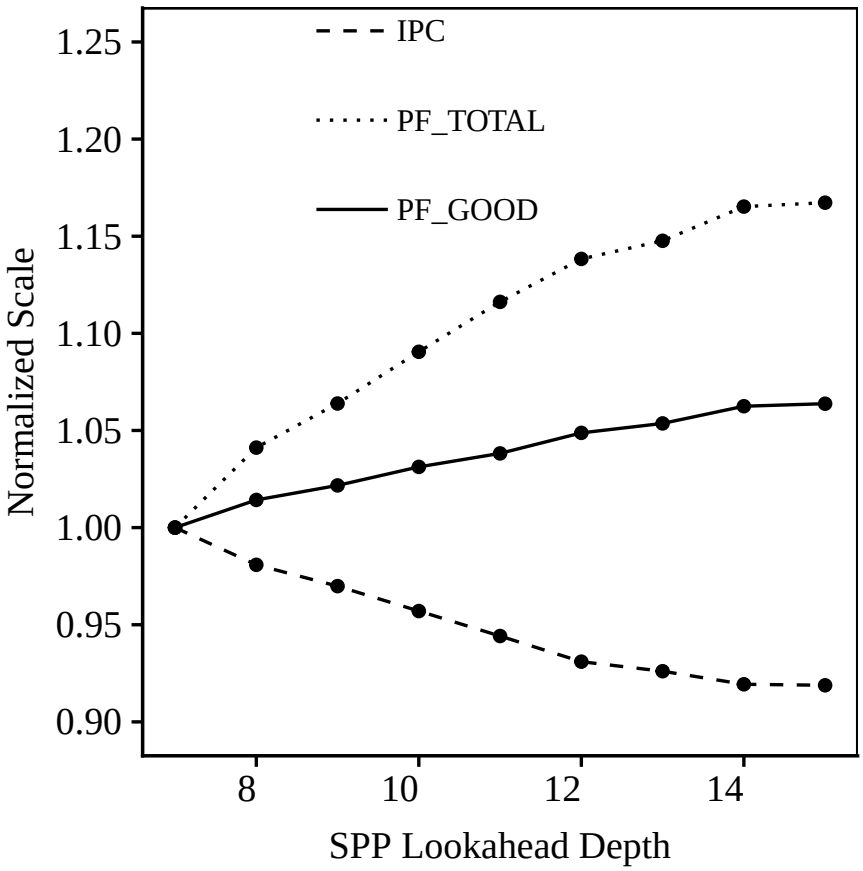
1. INTRODUCTION AND MOTIVATION

Processor and memory technologies have been developed with different goals in mind. While processor scaling has focused on speed improvements, memory scaling has primarily focused on increasing capacity. The difference in each technology's scaling has led to the Memory Wall [1] – the increasing gap between processor and memory performance. Data prefetching is one important technique that has been developed to minimize the effects of this trend. Data prefetching exploits the fact that in most applications, memory accesses have a repeating and predictable pattern. This presents an opportunity to speculatively fetch useful data from the DRAM into the cache hierarcy.

An ideal prefetching scheme would perfectly capture a program's memory access pattern, and then predict and pre-load the needed data into the processor's caches in a timely manner. Memory access patterns may be simple, such as accessing every item in an array with a for-loop, or very complex, such as chasing pointers through dynamically-allocated memory. Practical prefetchers face challenges not only in predicting what data will be useful in the future, but also in timing when that data should be prefetched, deciding which level of the cache hierarchy the data should be stored in, and deciding what data should be evicted from the caches to accommodate the prefetched data.

All prefetchers are designed around a fundamental trade-off between two important metrics: coverage and accuracy. Prefetcher coverage refers to the fraction of baseline cache misses that the prefetcher pulls into the cache prior to their reference. For example, if an application experiences 1,000 cache misses without a prefetcher, while 800 of those cache misses become hits with a prefetcher, then the prefetcher has 80% coverage for that application. Prefetcher accuracy refers to the fraction of prefetched cache lines that end up being used by the application. So if a prefetcher prefetches 1,200 cache lines, but only 800 of them are used by the application, then that prefetcher's accuracy is 66.7%.

Coverage and accuracy are generally at odds with one another, and as one metric improves, the other usually gets worse. For example, when an application accesses a new region of memory



(a) The impact of aggressive prefetching on performance for `603.bwaves_s`. The number of useful prefetches increases with aggressiveness slower than total prefetches, which wastes bandwidth and harms performance.

Figure 1.1: Performance vs Prefetch Aggressiveness

for the first time, a naïve prefetcher may predict that all data in that region will be used by the application. This will clearly result in 100% coverage for that region, but with possibly a very low accuracy. In fact, so much cache capacity and bandwidth may be wasted prefetching unused data that performance can ultimately be harmed by this strategy. At the other extreme, another prefetcher may be overly conservative and never prefetch anything, wasting no capacity or bandwidth, and achieving 0% prefetch coverage.

Figure 1.1 illustrates the above scenario. Here, I consider a state-of-the-art lookahead prefetcher – SPP [2]. Lookahead prefetchers such as SPP provide a mechanism to speculate an arbitrary number

of references ahead of the initial triggering access. In SPP, a throttling confidence threshold is then used to ensure that the lookahead stops when confidence falls too low to ensure that prefetches are accurate. In the figure, I iteratively re-tuned this threshold to allow the prefetcher to lookahead with depths varying from 7 to 15. The figure depicts the behavior of the `603.bwaves_s` SPEC CPU 2017 benchmark. The IPC, the total number of prefetches issued by the prefetcher (TOTAL_PF), and the actual useful predictions (GOOD_PF), all have been normalized to the lookahead depth of 7. As the lookahead depth increases, so do useful prefetches, and hence coverage. This coverage, however, comes at the cost of total prefetches increasing at an even higher rate. This leads to cache pollution and bandwidth contention, and leads to a reduction in IPC.

Therefore, a delicate balance between coverage and accuracy is required for a prefetcher to maximize its performance impact. Prefetchers are generally designed with internal mechanisms to monitor their accuracy, and throttling mechanisms that can be tuned for either coverage or accuracy. The more irregular an application's memory access pattern is, the more difficult it is to accurately predict every access, so a prefetcher will have to be tuned more toward coverage (and away from accuracy) in order to gain any benefit. This may be especially dangerous to do in the context of a multi-core processor, where overly aggressive prefetching in one core can waste shared resources, such as last-level cache (LLC) capacity, and off-chip bandwidth, impacting the performance of other cores [3].

From the preceeding discussuion, we can see that the core of prefetching lies in idenfying the memory access patterns and predicting the next access that a given pattern of accesses can lead to. Thus prefetching is a pattern recognition problem. We would also like to study the predictions and classify them as potentilly useful vs useless. This makes prefetching a classification problem too. Lastly, all the learning has to be on-line – happening in hardware in real-time, to make the prefetcher robust and diverse. This shows that prefetching has all the elements of utilizing a machine learing based aprroach.

Thesis Statement: In this thesis, I propose that machine learning based prefetch filtering can significantly improve performance and efficiency of existing prefetching schemes.

Proposed Solution: Here, I introduce Perceptron-based Prefetch Filtering (PPF) as an enhancement to existing state-of-the-art prefetchers, allowing them to speculate deeply to achieve high coverage while PPF filters out the inaccurate prefetches this deep speculation implies. PPF works by observing the stream of candidate prefetches generated by a prefetcher, and then rejects those that are predicted by the online-trained neural model to be inaccurate. The state-of-the-art prefetcher that I use to evaluate PPF in this work is the Signature Path Prefetcher (SPP) [2], however as described, PPF can be designed to benefit any prefetcher. In this design, PPF replaces SPP's existing confidence-based throttling mechanism, which itself was a highly tuned feature of that prefetcher. Because PPF is so much more effective at rejecting inaccurate prefetches than SPP's baseline mechanism, we are free to re-tune the rest of SPP's design around maximizing coverage. The result is an increase in both accuracy and coverage, and a notable increase in performance.

This work describes Perceptron-based Prefetch Filtering, explains its merits, offers analysis, and outlines the scope for future research. Its contributions are:

- An on-line neural model used for hardware data prefetching. Previous work in this area either relied on program semantics [4] or were application specific [5].
- Implementing PPF as a state-of-the-art prefetcher, giving a significant performance improvement compared to previous work. PPF adapts itself to shared resource constraints, leading to further increased performance in multi-core and bandwidth-constrained environments.
- A methodology for determining an appropriate set of features for prediction, regardless of the underlying prefetcher used. More details are explained in Section 4.1.5.

In a single core configuration, PPF increases performance by 3.78% compared to the underlying prefetcher, SPP. In a multi-core system running a mixes of memory intensive SPEC CPU 2017 traces, PPF sees an improvement of 11.4% over SPP for a 4-core system, and 9.65% for an 8-core system.

2. BACKGROUND AND RELATED WORK

In this chapter, I discuss the most closely related work to this proposed technique. A major portion of this chapter discusses prior work in the domain of hardware prefetching, including the Signature Path Prefetcher, which acts as the underlying prefetcher in the implementation done in this work. The remaining part gives a conext on application of machine learning for hardware predictions.

2.1 Background

The idea of prefetching begins with Jouppi's *Instruction Stream Buffers* [6]. Two of the most fundamental prefetchers are the Next-n Lines [7] and the Stride based [8, 9, 10] prefetchers. Both capture regular memory access patterns with low overhead. The Next-n Lines prefetcher queues prefetches for the next n blocks after any given miss, expecting that the principle of spatial locality will hold and those cache blocks after a missed block are likely to be used in the future. The stride prefetcher identifies strided reference patterns in programs based on past behavior of missing loads. When a load misses, cache blocks ahead of that miss are fetched in the pattern following the previous behavior in the hope of avoiding future misses. Without further knowledge about temporal locality and application characteristics, these prefetchers cannot do more than detecting and prefetching regular memory access patterns with limited spatial locality.

Modern prefetching mechanisms are more sophisticated as they look into past memory behavior [11, 12], locality [13, 14, 15, 16, 17, 18], control-flow speculation [19, 20], and other other aspects to detect complex memory access patterns.

2.1.1 Spatial Prefetchers

Spatial prefetchers include such well-understood examples as the next-line (or next-n-line) prefetcher, and the stream prefetcher, and are distinguished by prefetching data without regard for the order in which the data will be accessed. In addition to these simpler examples, Somogyi *et al.* propose Spatial Memory Streaming (SMS) [14]. SMS works by learning the spatial footprint

of all data used by a program within a region of memory around a given missing load, and when the load that causes an new miss elsewhere, the same spatial footprint is prefetched. Ishii *et al.* propose the Access Map Pattern Matching prefetcher (AMPM) [12], which creates a map of all accessed lines within a region of memory, and then analyzes this map on every access to see if any fixed-stride pattern can be identified and prefetched that is centered on the current access. DRAM-Aware AMPM (DA-AMPM) [21] is a variant of AMPM that delays some prefetches so they can be issued together with others in the same DRAM row, increasing bandwidth utilization. Pugsley *et al.* propose the Sandbox Prefetcher [22], which analyzes candidate fixed-offset prefetchers in a sandboxed environment to determine which is most suitable for the current program phase. Michaud proposes the Best-Offset Prefetcher [23], which determines the optimal offset to prefetch while considering memory latency and prefetch timeliness.

2.1.2 Lookahead Prefetchers

Unlike spatial prefetchers, lookahead prefetchers take program order into account when they make predictions. Shevgoor *et al.* propose the Variable Length Delta Prefetcher (VLDP) [24], which correlates histories of deltas between cache line accesses within memory pages with the next delta within that page. SPP [2] and KPC's prefetching component [25] are more recent examples of lookahead prefetchers. They try to predict not only what data will be used in the future, but also the precise order in which the data will be used, within a given page. Predictions made by lookahead prefetchers can be fed back into their prediction mechanisms to predict even further down a speculative path of memory acesses. These prefetchers can also generalize their learned patterns from one page, and use those patterns to make predictions in other pages.

2.1.3 Managing Prefetched Data

A low-accuracy aggressive prefetcher can significantly harm performance. To minimize interference from prefetching, Wu *et al.* propose PACMan [26], a prefetch-aware cache management policy. PACMan dedicates some LLC sets to each of three competing policies that treat demand and prefetch requests differently, using the policy in the rest of the cache that shows the fewest

misses. Seshadri *et al.* propose ICP [27], which demotes a prefetch to the lowest reuse priority on a demand hit, based on the observation that most prefetches are dead after their first hit. To address prefetcher-caused cache pollution, it also uses a variation of EAF [28] to track prefetching accuracy, and inserts only accurate prefetches to the higher priority position in the LRU stack. Jain *et al.* propose Harmony [29] to accomodate prefetches in their MIN algorithm-inspired Hawkeye cache management system.

2.1.4 Machine Learning for Prefetching

Peled *et al.* introduce interesting ideas for on-line Reinforcement Learning and dynamically scaling the magnitude of feedback given to the baseline prefetcher [4]. The prefetcher relies on compiler support to receive features and build the context. Liao *et al.* focus on prefetching for data center applications [5]. They use offline machine learning algorithms such as SVMs and logistic regression to do a parametric search for an optimal prefetcher configuration.

2.2 Perceptron Learning in Computer Architecture

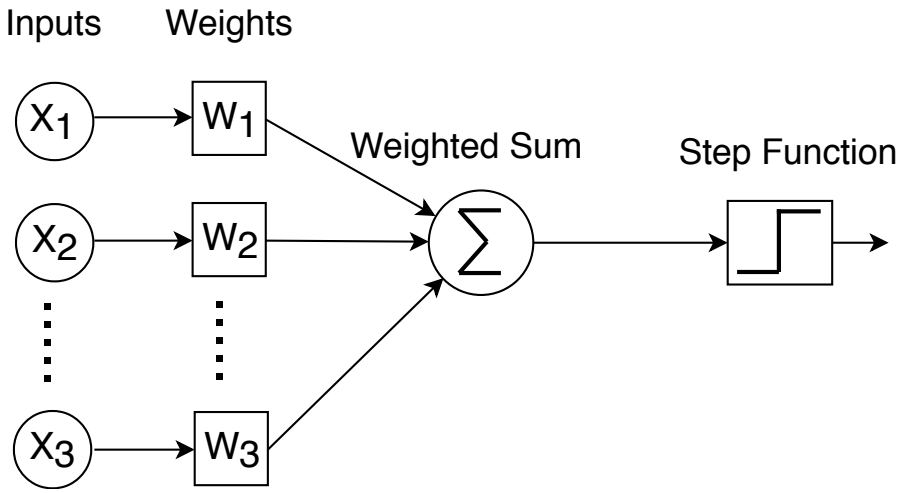


Figure 2.1: The Basic Perceptron Model

2.2.1 Perceptron Training

Perceptron learning is the most basic form of a machine learning based prediction mechanism. As shown in the figure 2.1, a perceptron is a light weight structure that can add contributions from disparate pieces of information, called as a feature or an attribute, into a meaningful sum.

Mathematically, a perceptron is modeled as a dot product of the signed weights with the vector of input features. These weights are learned through online training. The sum obtained from the dot product depicts the extent of confidence in predicting the final decision. In a way, training the perceptron weights helps capture the correlation of the decision prediction with the input features.

There are two components to the abstract perceptron learning algorithm:

- To predict true or false, a vector of signed weights is chosen according to some criteria. The dot product of the weights and input vectors, called y_{out}, is computed. If y_{out} exceeds some threshold, the prediction is true, otherwise it is false.

- To update the weights after the true outcome of the predicted event is known, we first consider the value of y_{out}. If the prediction was correct and $|y_{\text{out}}|$ exceeds a threshold θ, then the weights remain unchanged. Otherwise, the inputs are used to update the corresponding weights. If there is positive correlation between the input and the outcome of the event, the corresponding weight is incremented; otherwise, it is decremented. Over time, the weights are proportional to the probability that the outcome of the event is true in the context of the input and the criterion for choosing that weight. The weights saturate at maximum and minimum values so they will fit in a fixed bit width.

2.2.2 Hashed Perceptrons for Branch Prediction

Perceptron learning for microarchitectural prediction was introduced for branch prediction [30]. This predictor uses a version of microarchitectural perceptron prediction known as the "hashed perceptron" organization [31]. As an abstract idea, a hashed perceptron predictor hashes several different features into values that index several distinct tables. Small integer weights are read out from the tables and summed. If the sum exceeds some threshold, a positive prediction is made, *e.g.*

"predict branch taken" or "allow the prefetch." Otherwise, a negative prediction is made. Once the ground truth is known, the weights corredponding to the prediction are incremented if the outcome was positive, or decremented if it was negative. This update only occurs if the prediction was incorrect or if the magnitude of the sum failed to exceed a threshold. Beyond branch prediction, perceptron learning has been applied to last-level cache reuse prediction [32, 33]. In this work, I apply it for the first time to prefetch filtering.

2.2.3 Perceptrons in Cache Management

In addition to branch prediction [30], perceptron-based learning has been applied to the area of cache management. Teran *et al.* propose using perceptrons to predict cache line resuse, bypass, and replacement [32]. Perceptron Learning trains weights selected by hashes of multiple features, including the PC of the memory access instruction, some other recent PCs, and two different shifts of the tag of the referenced block. These features are used to index into weight tables, and the weights are then thresholded to generate a prediction. When a block from one of a few sampled sets [34] is reused or evicted, the corresponding weights are decremented or incremented, according to the perceptron learning rule. Multiperspective Reuse Prediction [33] improves on Perceptron Learning by contributing many new features.

2.3 Baseline Prefetcher: SPP

Kim *et. al.* proposed Signature Path Prefetcher (SPP) [2], a confidence-based lookahead prefetcher. SPP creates a signature associated with a page address by compressing the history of accesses. By correlating the signature with future likely delta patterns, SPP learns both simple and complicated memory access patterns quickly. While the basic idea of perceptron based prefetch filtering is applicable to any lookahead prefetcher, this work develops a practical implementation of the proposed perceptron prefetch filter using SPP as the baseline. Here I describe the basic architecture of SPP.

Signature Table: As shown on the left side of Figure 2.2, the Signature Table keeps track of 256 most recently accessed pages. It is meant to capture memory access patterns within a page

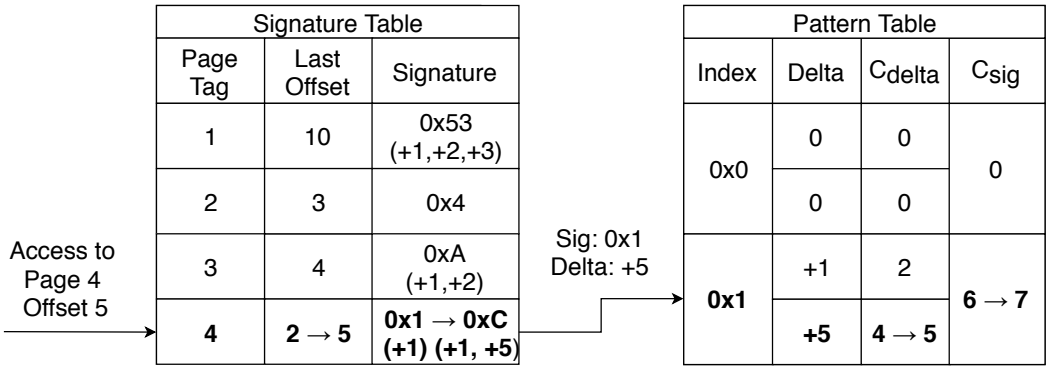


Figure 2.2: SPP Data-path Flow

boundary. SPP indexes into an entry of the Signature Table using the page number. For each entry corresponding to a page, the table stores a 'last block offset' and an 'old signature'. Last block offset is the block offset of the last memory access of that given page. The block offset is calculated with respect to the page boundary. The signature is a 12-bit compressed representation of the past few memory accesses for that page. The signature is calculated as:

$$NewSignature = (\,OldSignature << 3bits\,)\ \ XOR\ \ (\,Delta\,)$$

Delta is the numerical difference between the block offset of the current and the previous memory access. In case a matching page entry is found, the stored signature retrieved and used to index into the Pattern Table. This process is illustrated in Figure 2.2.

Pattern Table: The Pattern Table, shown on the right side in Figure 2.2 is indexed by the signature generated from the Signature Table. Pattern Table holds predicted delta patterns and their confidence estimates. Each entry indexed by the signature holds up to 4 unique delta predictions.

Lookahead Prefetching: On each trigger, SPP goes down the program speculation path using its own prefetch suggestion. Using the current prefetch as a starting point, it re-accesses the Pattern Table to generate further prefetches. As illustrated in Figure 2.3, it repeats the cycle of accessing Pattern Table and updating the signature based on highest confidence prefetch from the last iteration.

The iteration counter on which SPP manages to predict prefetch entries in the lookahead manner is characterized as its 'depth'. While doing so, SPP also keeps compounding the confidence in each depth. Thus as depth increases, overall confidence keeps decreasing.

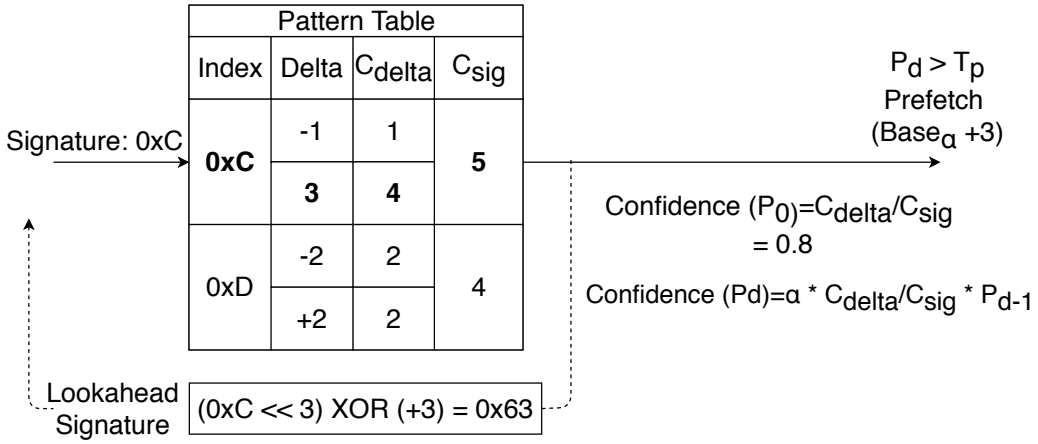


Figure 2.3: SPP Architecture

Confidence Tracking: As shown in Figure 2.3, the Pattern Table keeps a count of hits to each signature through a counter C_{sig}. The number of hits for a given delta per signature are tracked using a counter C_{delta}. The confidence for a given delta is approximated through $C_d = C_{delta} / C_{sig}$. When SPP enters into a lookahead mode, the path confidence P_d is given as: $P_d = \alpha \, . \, C_d \, . \, P_{d-1}$ Here α represents the global accuracy, the ratio of number of prefetches which led to a demand hit to the total number of prefetches recommended. The range of α is [0,1]. d is the lookahead depth. For $d = 1$ SPP is in non-speculative mode and $P_0 = 1$. The final P_d is thresholded against prefetch threshold (T_p) to reject the low confidence suggestions and then against a numerically bigger fill threshold (T_f) to decide whether to send the prefetch to L2 Cache (high confidence prefetch) or Last Level Cache (low confidence prefetch). The two thresholds were empirically set to 25 and 90 respectively, on the scale of 0 to 100.

2.4 Case for an On-line Filter

As was noted in Figure 1.1, aggressive lookahead prefetching, if done without any accuracy check, can harm the performance of the system. As the figure shows, aggressive lookahead and its accompanied loss of accuracy degrades performance by almost 9%. This is despite a growing number of useful prefetches generated by the prefetcher. Thus, we need a mechanism that is orthogonal to the underlying prefetching scheme and can be used to prune out the useful prefetches from the useless ones.

Moreover, the on-line confidence mechanism used by most prefetchers is very rudimentary. For example, SPP's internal confidence mechanism is based on taking the ratio $C_d = C_{delta} / C_{sig}$. This confidence was used to make the decision of whether to prefetch or not to prefetch and which level to prefetch. While this approximation was shown to work in the original implementation, I believe that a better form of generalized on-line decision making was possible. Hence, it was necessary to build a robust and adaptable learning mechanism to accept / reject the prefetch suggestions and to decide the fill level (L2 Cache vs Last Level Cache). Thus, in this thesis, I introduce an independent on-line perceptron based filtering mechanism.

3. PERCEPTRON BASED PREFETCH FILTERING

This chapter describes the details of the proposed idea - Perceptron Based Prefetch Filtering (PPF). The first section talks about the general architecture of the prefetch filter and how it can be developed for any underlying prefetcher. The second section consists of the practical implementation of PPF, using SPP as the baseline prefetcher. It also describes how PPF can be tweaked to work in a tightly knit manner with the underlying prefetcher.

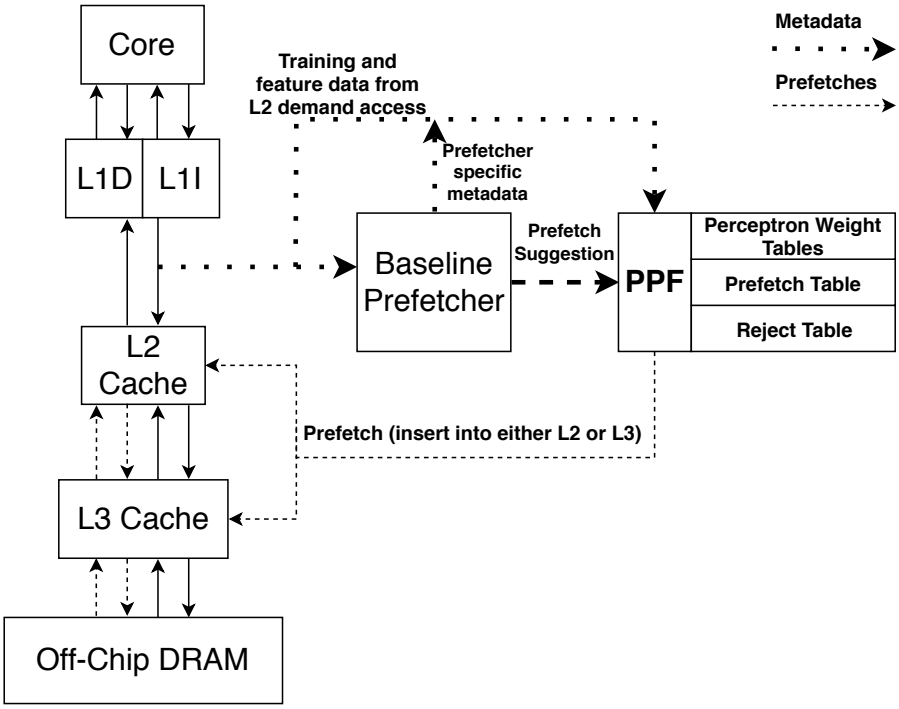


Figure 3.1: PPF Architecture in the Memory Hierarchy

3.1 PPF Design and Architecture

It can be beneficial to allow a prefetcher to speculate as deeply as possible. Often, some useful prefetches are generated long after the confidence of the prefetcher has fallen below the point at which performance degrades due to the increase of inaccurate prefetches. In order to allow deep

speculation in the prefetcher, however, inaccurate prefetches must be filtered out. I propose to leverage perceptron-based learning as a mechanism to differentiate between potentially useful deeply speculated prefetches and likely not-useful ones. The Perceptron Prefetch Filter (PPF) is placed between the prefetcher and the prefetch insertion queue, as illustrated in Figure 3.1, to prevent not-useful prefetches from polluting the higher levels of the memory hierarchy.

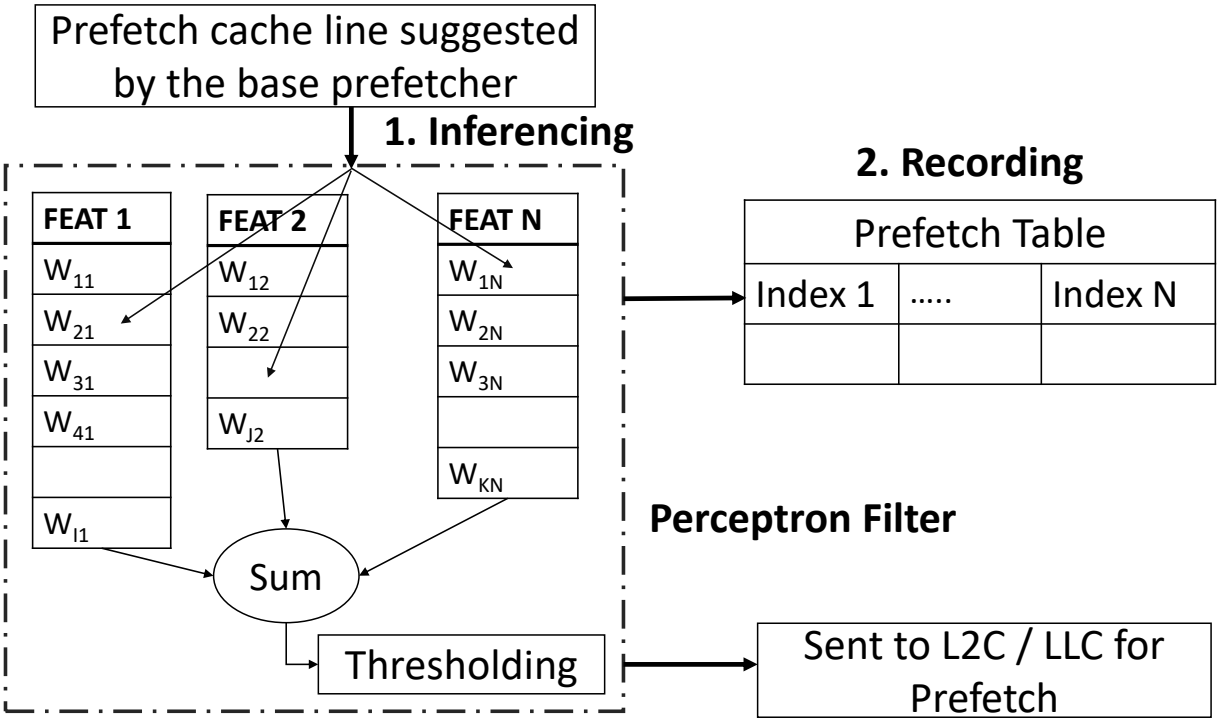
Perceptron learning is a light-weight mechanism to pull together disparate forms of information and synthesize a decision from them. This work considers a number of features corresponding to a prefetch, such as speculation depth, page address and offset, and uses this information as the inputs to the perceptron-based filter in order to predict the usefulness of a prefetch. Here, I discuss the design of my proposed perceptron prefetch filter (PPF). PPF enhances a base prefetcher by filtering out predicted unused prefetches. PPF is a generalized prefetch filtering mechanism that may be adapted to any prefetcher with appropriate feature selection and modifications described below.

3.1.1 The Perceptron Filter

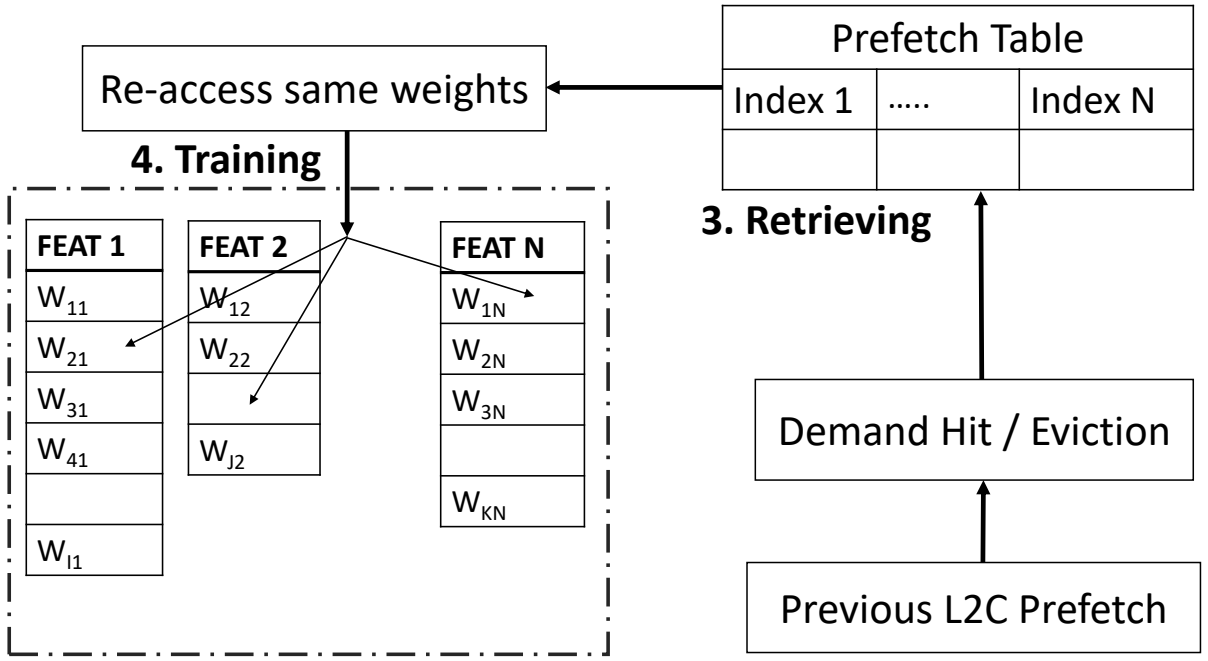
Figure 3.2 shows the microarchitecture of PPF, as well as the steps required to filter out not-useful prefetches. The perceptron filter is organized as a set of tables, where each entry in the tables holds a *weight*. For a configuration of PPF using N number of features, N different tables of weights are needed. Each feature is used to index into a distinct table. The number of entries of each table varies according to the corresponding feature, hence, different number of bits are needed to index different tables. Each weight is a 5-bit saturating counter ranging from -16 to +15. I found that having 5-bit weights provides a good trade-off between accuracy and area. A detailed explanation of the storage overhead of PPF can be found in Section 4.1.6.

Inferencing

The base prefetcher is triggered on every demand access to the L2 Cache, as seen in Figure 3.1. At this point, it has the opportunity to trigger a prefetch. If it does so, it will also need to decide how many cache blocks to prefetch. These blocks can be either placed in the L2 or L3 cache according to the confidence of the prefetching mechanism. Once the base prefetcher is triggered, the suggested prefetch candidates are fed to the perceptron filter to determine the usefulness of these prefetches.



(a) Prediction



(b) Update

Figure 3.2: PPF Data Path and Operation

The filter ultimately decides whether to issue the prefetch suggestions of the base prefetcher. As shown in step 1 of Figure 3.2(a), to make the decision, each feature corresponding to a suggested prefetch is used to index a table and all the corresponding weights are summed. The sum denotes the confidence value for the suggested prefetch, and is thresholded against two different values: τ_{hi} and τ_{lo}.

Prefetches whose sum exceeds τ_{hi} are placed into the L2 cache. The higher confidence value hints the prefetch would be useful and should be prioritized. A prefetch for which the features result in a confidence value between τ_{lo} and τ_{hi} is allocated in the larger LLC, as the filter is moderately confident of the future reuse of the cache block, but not enough to possible pollute a significantly smaller L2. Suggested prefetches for which the features lead to a confidence value lower than τ_{lo} are not prefetched, as the low confidence value represents that the perceptron learned that a similar set of features are associated with non-useful prefetches.

Recording

As shown in step 2 of Figure 3.2(a) the prefetches that make it through the inference stage are recorded in the "Prefetch Table". The prefetch table is a 1,024-entry, direct mapped structure that contains all metadata required to re-index the perceptron entries for training.

In addition to the prefetch table mentioned above, PPF also maintains a 1,024-entry direct-mapped "Reject Table." If a prefetch suggestion is rejected by the perceptron layer, it is logged into the reject table. The table is used to train the perceptron to avoid false negatives *i.e.*, cases where the prediction suggested to reject the prefetch but the prefetch could have proven to be useful based on the observed demand accesses to the L2.

Feedback and Data Retrieval

As depicted in step 3, when there is an eviction or a demand access to the L2, training for both the base prefetcher and the filter mechanism is triggered. The address of the cache block that triggered training is used to index both the Prefetch and Reject tables. 10 bits of the address are used to index tables, and another 6 bits to perform tag matching. Once the contents of the matching entry have been retrieved, the corresponding features are used to index into the tables of weights again.

Training

As can be seen in step 4, the address from the demand request triggering the training is looked up in both tables. If the address is in the prefetch table and marked as valid, this hints the previous prediction was correct and this is a useful prefetch. The prefetcher computes the sum of the corresponding weights. If the sum falls below a specific threshold, training occurs and the corresponding weights are adjusted accordingly. These thresholds are introduced in order to avoid over-training, helping the filter adapt quickly to changes in memory behavior. These thresholds are referred to as θ_p and θ_n, respectively for the positive and negative values of training saturation.

On a cache block eviction, the prefetching logic looks up the corresponding address in the prefetch table. If there is a valid entry with this address, the filter made a misprediction. The block was allocated in the L2 with a prefetch request that the filter should have categorized as a useless prefetch. Thus, the corresponding features of the prefetch request are used to re-index the tables of weights, and those weights are adjusted accordingly.

Parallel to accessing the prefetch table, on a demand access, the reject table is accessed. Before the demand access triggers the next set of prefetches, the reject table is checked for a valid entry. A hit means that the corresponding cache block was initially suggested by the base prefetcher, but wrongly rejected by the perceptron filter. The perceptron filter learns from this and makes use of the corresponding features associated to the original prefetch request, which are stored in the reject table, to index the weights tables and adjust the weights accordingly. The implementation of the reject table, allows the prefetcher to capture the information of prefetches that were rejected, and that can be used to further optimize the overall prefetching mechanism.

3.1.2 Optimizing PPF for a Given Prefetcher

The above discussion of PPF shows that it is highly modular and can be adapted to be used over any base prefetcher for increased prefetch accuracy. As a first step, all the prefetch candidates of the prefetcher have to pass through the perceptron filter. If qualified, the metadata for perceptron indexing has to be stored. Next, when the feedback of a prior prefetch is available in form of a subsequent demand hit or cache eviction, the stored metadata needs to be retrieved to update the

state of the perceptrons.

In general, PPF can be adapted to a new base prefetcher with only a few modifications:

Making Base Prefetcher More Aggressive: By tuning down any internal thresholds or throttling mechanisms to increase its aggressiveness.

Inferencing and Storing: All prefetch recommendations are tested using the perceptron inferencing algorithm. The perceptron's output, *true* or *false*, should be saved appropriately, along with all metadata required for perceptron indexing.

Retrieving and Training: When feedback for a prefetch becomes available, the previously stored metadata can be used to re-index into the perceptron entries and increment or decrement the weights.

Feature Selection: Perceptrons essentially integrate contributions from different features to get a single sum representing the final confidence. Thus, perceptron learning can only be as good as the set of features chosen. Interestingly, this is what makes perceptron learning scalable, as it can easily learn to incorporate newer information in the form of new features. Some of the features developed here use information derived directly from program execution, agnostic to the baseline prefetcher. Beyond that, the feature set can be expanded to convey any useful information or metadata available in the baseline prefetcher.

Using Metadata from the Prefetcher: Some of the internal counters specific to the underlying prefetcher can be suitable candidates for the perceptron features. To make sure that the perceptron layer sees that, the relevant metadata must be exported from the prefetcher to PPF. This way, PPF can be optimized to work tightly-knit with the base prefetcher.

3.2 PPF Implementation Using SPP

This section describes a case study implementation of PPF and the range of features that are used to determine the usefulness of prefetches. Here, I have selected SPP as the base engine prefetcher.

3.2.1 Changes Made to SPP

To modify the SPP design to suit my scheme, the following changes were made:

Exporting Features from SPP: PPF uses the metadata specific to SPP to build some of the perceptron features. These include the lookahead depth, signature and the confidence counter. These features were made visible to PPF.

Original Thresholds Discarded: In PPF, the perceptron sum is used to decide whether to prefetch or not, and the fill-level in case of prefetch. Thus, the confidence thresholds used by SPP – T_f and T_p are no longer needed to throttle the prefetcher and can be discarded.

3.2.2 Features used by Perceptron

All the features used by PPF can be derived from the information available in the L2 Cache access stream or are taken as metadata derived from the baseline prefetcher. My feature selection involved searching over a large space of relevant perceptron features. Note that part of the process of tuning PPF to a specific prefetcher involves examining the available metadata in the prefetcher itself, and thus PPF is attuned to the baseline prefetcher's design. Using the statistical methodology outlined in Section 4.1.5, I pruned the feature set to a minimal yet relevant set of features. When implemented with SPP as the baseline prefetcher, the following nine features were used by PPF:

- **Physical Address**: Here I use the lower bits of the physical address of the demand access that triggers the prefetch. This address corresponds to a stream of accesses that SPP and the PPF have seen before. Therefore, PPF will correlate the past behavior of this address to the prefetch outcome.

- **Cache Line** and **Page Address**: These two separate features are derived from the base address that triggered the prefetch, as follows: $base\_addr >> LOG2\_BLOCK\_SIZE$ and

$base\_addr >> LOG2\_PAGE\_SIZE$ respectively. The idea behind using three different shifted versions of the same feature is that it allows the filter to focus its examination in more detail on different aspects of the address than with a single version. It also helps give more importance to the overlapping bits and lesser importance to most and least significance bits. This approach can also eliminate destructive interference that can be caused by directly folding the address bits into half.

- **Program Counter XOR Depth**: The PC is for the instruction that triggered the prefetch chain. Depth refers to the iteration count of the lookahead stages. It was seen that by itself, the PC is not a good basis for filtering a lookahead prefetcher, as all the prefetches with depth >= 1 are aliased into the same PC, which will not be the PC of the eventual actual demand access. This feature, on the other hand, resolves a PC into a different value for each lookahead depth of prefetch speculation, giving a more accurate correlation in lookahead cases. This is akin to the concept of Virtual Program Counters [35] introduced by Kim *et. al.* for indirect branch prediction.

- **PC_1 XOR PC_2»1 XOR PC_3»2**: Here PC_i refers to the last i^{th} PC before the instruction that triggered the current prefetch. Hashing together the last three PCs informs PPF about the path that led to the current demand access and helps capture and branching information of the current basic block. PCs are shifted in the increasing order of history before being hashed together. This is done to avoid the resultant value of zero when 2 or more PCs are the same. Additionally, blurring the information as it gets older allows us to get a wider and yet more approximate look into the program's history.

- **Program Counter XOR Delta**: This feature tells us if a given PC favors particular value(s) of delta. As noted earlier, while the PC alone does not convey useful information, this hash resolves the PC into different values based on the tendency of that PC to favor a certain delta. Thus, the dynamic nature of different instances of the same memory access instruction is captured here.

- **Confidence**: The confidence, on a scale of 0 to 100, used to throttle lookahead depth in the original SPP design. While the original confidence does not directly make the decision to prefetch, PPF correlates it to the correctness of a proposed prefetch. While the original SPP may have dismissed a prefetch due to running further into speculation, PPF can use the original confidence as indicator of not only when prefetches become less confident, but also how likely a low confident speculation is correct in the context of other features.

- **Page Address XOR Confidence**: This feature scores the tendency of each page to be prefetch friendly or prefetch averse. It helps resolve a page into different entries depending on its confidence for prefetching, which can vary during phases of a program execution.

- **Current Signature XOR Delta**: Recall from the discussion of SPP in Section 2.3 that the new signature is generated using the old signature and the current block delta. The result of this feature is the next signature that is predicted to be accessed based on the delta predicted by SPP. While "Current Signature XOR Delta" is not the exact formula for generating the future signature, it gives an approximate idea of the path that the combination of these two values can lead to.

As can be noted above, some composite features are derived from simple hashing (XOR) of two primary features. There is always a question of usefulness of such composite features and the new information added. I justify the choice of each feature by quantifying the contribution made towards predicting prefetch behavior, in Section 4.1.5. Finally, as noted above, each feature indexes into its independent entry of perceptron weights.

4. METHODOLOGY AND RESULTS

4.1 Methodology

4.1.1 Performance Model

I use the ChampSim [36] simulator for the evaluation of PPF against prior techniques. ChampSim is an enhanced version of the framework that was used for the 2nd Data Prefetch Championship (DPC-2) [37], also used in the 2nd Cache Replacement Competition (CRC2) [38]. I model 1-core, 4-core, and 8-core out-of-order machines. The details of the configuration parameters are summarized in Table 4.1.

The block size is fixed at 64 bytes. Prefetching is only triggered upon L2 cache demand accesses but could be directed to the L2 or last-level cache. No L1 data level prefetching is done. The LRU replacement policy is used on all levels of cache hierarchies. Branch prediction is done using the perceptron branch predictor [30]. The page size is 4KB. ChampSim operates all the prefetchers strictly in the physical address space.

4.1.2 Testing Under Additional Memory Constraints

The default single-core configuration simulates a 2MB LLC and a single channel DRAM with 12.8GB/s bandwidth. I extend the simulations to include memory constraints introduced in DPC-2. Specifically I look at the following two variations: Low Bandwidth DRAM, where DRAM bandwidth is limited to 3.2 GB/s, and small LLC, where the LLC size is reduced to 512 KB. All the multi-core simulations are only done in the default configuration.

4.1.3 Workloads

I use all the 20 workloads available in the SPEC CPU 2017 suite [39]. Using the SimPoint [40] methodology, I identified 95 different program segments of 1 Billion instructions each.

Single-core performance: For single-core simulations, I use the first 200 million instructions to warm-up the microarchitectural structures and the next 1 billion instructions to do detailed

Block	Configuration
CPU Core	1-8 Cores, 4 GHz 256 entry ROB, 4-wide
Private L1 DCache	32 KB, 8-way, 4 cycles 8 MSHRs, LRU
Private L2 Cache	256 KB, 8-way, 8 cycles 16 MSHRs, LRU, Non-inclusive
Shared LLC	2MB/core, 16-way, 12 cycles 32 MSHRs, LRU, Non-inclusive
DRAM	4 GB 1-Channel (single-core) 8 GB 2-Channels (multi-core) 64-bit channel, 1600MT/s

Table 4.1: Simulation Parameters

simulations and collect run-time statistics. I report the IPC speedup over the baseline of no prefetching. The final numbers reported are the geometric mean of the weighted mean speedup achieved per application using the SimPoint methodology.

Multi-core performance: For multi-application workloads, I generate 100 random mixes and another 100 mixes from the memory intensive subset of SPEC CPU 2017. For 4-core workloads, 200 Million instructions are used for warm-up and additional 1 Billion instruction simulated for collecting statistics. Each CPU keeps executing its workload till the last CPU completes one billion instructions after warm-up. For collecting IPC and other data, only the first billion instructions are considered as the region of interest.

Here I report the weighted speedup normalized to baseline *i.e.*, no prefetching. For each of the workloads running on a particular core of the 4-core 8 MB LLC system, I compute IPC_i. I then find the $IPC\_isolated_i$ of the same workload running in isolated 1-core 8 MB LLC environment. Then I calculate the total weighted-IPC for a given workload mix as Σ (IPC_i / $IPC\_isolated_i$). For each of the 100 workload-mix, the sum obtained is normalized to the weighted-IPC calculated similarly for baseline case *i.e.*, no prefetching, to get the weighted-IPC-speedup. Finally the geometric mean of these 100 weighted-IPC-speedup is reported as the effective speedup obtained by the prefetching scheme.

I repeat the same process for 8-core workloads, correspondingly with 16MB LLC. The only difference is that 20 million warm-up instructions and 100 million full instructions are executed. This is done so as to keep the simulation run-time within reasonable limits as a single 8-core mix takes up to 3 days to simulate one billion instructions.

Validation: I cross-validated the PPF model using SPEC CPU 2006 [41] and CloudSuite [42] benchmarks. For single-core SPEC CPU 2006, I developed 94 simpoints spread across all the 29 applications. For multi-core, I followed the same methodology as SPEC CPU 2017. For CloudSuite, I used the traces made available for the 2nd Cache Replacement Competition (CRC-2) [38]. The traces include four 4-core applications with six distinct phases per application.

In total, I used 285 traces representing workloads across 53 applications. Throughout the thesis, I consider memory intensive subset as the applications with SimPoint weighted LLC MPKI > 1. This includes 11 out of 20 SPEC CPU 2017 applications. For SPEC CPU 2006, this includes 16 out of 29 applications.

4.1.4 Prefetchers Simulated

I compare PPF against three of the latest, state of the art hardware-only prefetchers: Best Offset Prefetcher (BOP), DRAM Aware - Access Map Pattern Matching (DA-AMPM) [21] and Signature Path Prefetcher (SPP). BOP was the winner of 2nd Data Prefetching Championship. DA-AMPM is the enhanced version of AMPM, modified to account for DRAM row buffer locality. SPP has been shown to outperform BOP on SPEC CPU 2006 traces [2]. For each of these, I compare their speedups taking the no prefetching case as the baseline.

4.1.5 Developing Features for PPF

This section describes the intuition and analysis that went behind developing the perceptron features. As noted earlier, I developed a set of nine features that allow the perceptron layer to correlate prefetching decision with the program behavior. To study the correlation across each feature, I statistically examine the perceptron weights and try to interpret their distribution.

Global Pearson's Correlation: Here I examine the perceptron weights at the end of all trace

execution by which time the weights have settled to steady values. The weights obtained from running all the SPEC CPU 2017 traces are concatenated. Features with a bulk of their perceptron weights concentrated around 0 or small magnitude numbers show a weak correlation with the prefetching outcome. On the other hand, features with most of the weights saturated around highest value (+15) show a high positive correlation and the features with weights close to the lowest value (-16) show a strong negative correlation.

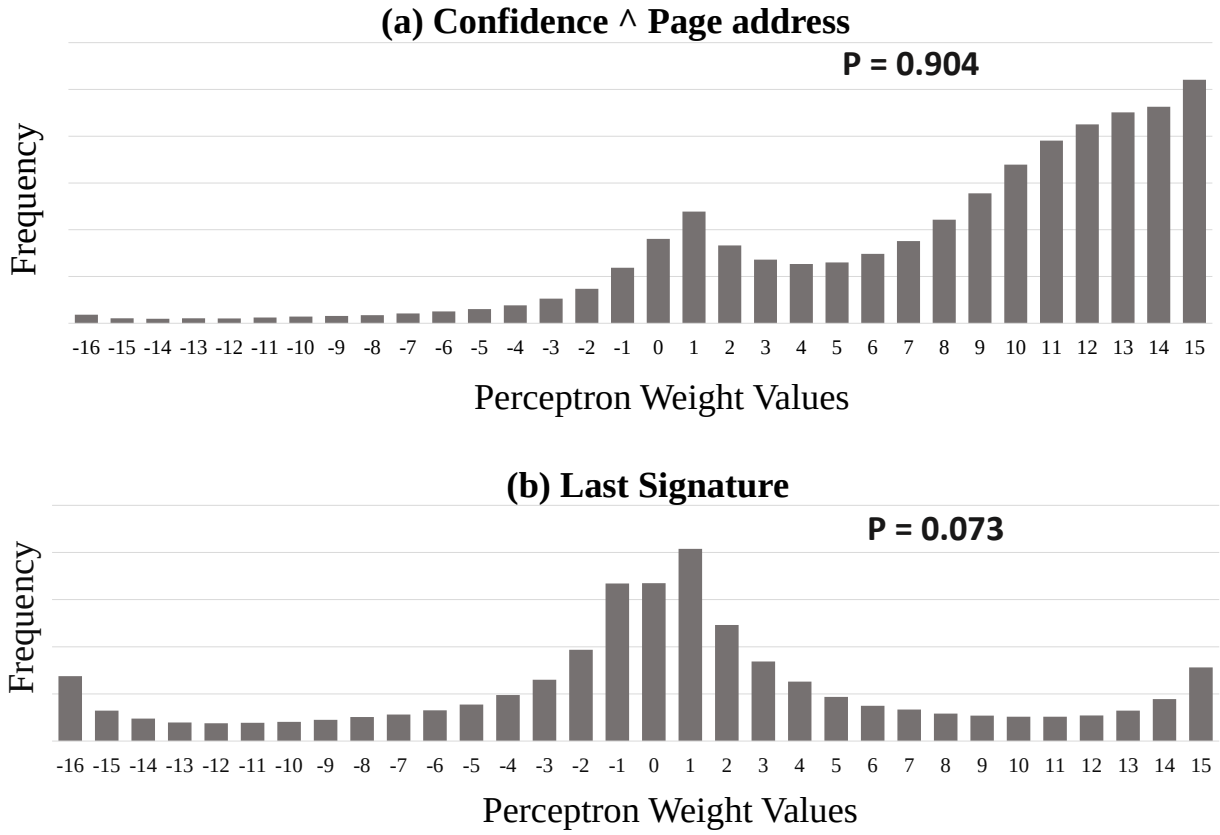


Figure 4.1: Distribution of Trained Weights

I plot a histogram for each feature depicting weights distribution from -16 to +15 and generate the Pearson's correlation factor for that feature. Pearson's factor is a numerical measure ranging from -1 to 1 of the degree of linear correlation between two variables. The magnitude of Pearson's

factor gives the extent of correlation and the sign indicates whether it is a positive correlation or a negative correlation. Values close to 0 suggest a low correlation while a value of +1/-1 suggests a perfectly linear positive / negative correlation respectively.

As a part of the perceptron feature selection methodology, I explored a wide variety of features to begin with. Features with a low Pearson's coefficient were rejected as they didn't provide much useful correlation. Figure 4.1 depicts the histogram distribution of trained weights for two features. The first feature, *Confidence XOR Page address* has the highest observed P-value, hence was retained. On the other hand, the second feature, *Last Signature* did not provide any meaningful correlation and hence was rejected. This is visible from the bulk of its trained weights settling to near zero values.

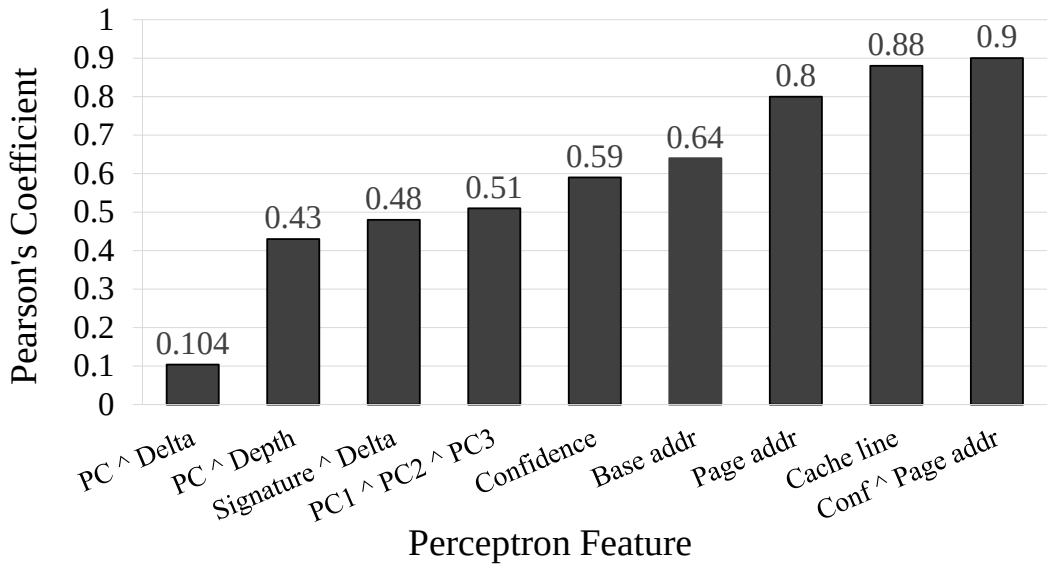


Figure 4.2: P-Values for all Features

Figure 4.2 shows all the features which are finally used, arranged in the increasing order of their Pearson's factor. As can be seen 5 out of the 9 features provide a moderate to high correlation, with

the magnitude of P-value > 0.6. The single most important feature, *Confidence XOR Page address* helps provide a correlation to prefetch outcome with a factor of 0.90.

Per Trace Correlation: Another important way to look at the perceptron features is to see how much their contribution varies across the traces. Here I give special attention to features with low overall P-values. Figure 4.3 shows the variation of P-values for three features : *PC XOR Delta*, *Signature XOR Delta* and *PC XOR Depth*; across all the SPEC CPU 2017 traces. For simplicity, the traces are arranged in an increasing order of contribution made by the feature. It can be seen that even features with a low overall correlation provide useful correlation (magnitude > 0.5) for a significant number of traces. This study motivated me to choose *PC XOR DELTA* over *Last Signature* as it provided useful correlation in at least some of the traces.

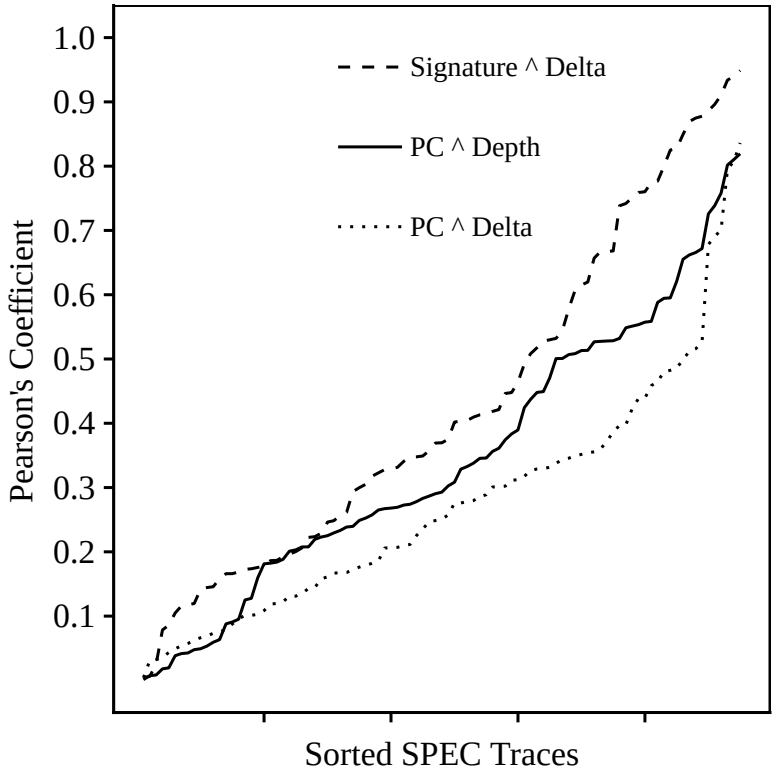


Figure 4.3: P-value Variation across Traces for selected Features

Trimming Features Using Cross-correlation: Beside providing interesting insights into prefetching behavior, P-value can also be used for feature selection and prefetcher tuning.

As I examined correlation of each feature with the final outcome, I also studied correlation between the features. I used the above methodology to eliminate features providing little information that has already been captured in other features.

I initially came up a with set of 23 features. By studying cross correlation of each of these features against others in a 23×23 matrix, I identified pairs of features with correlation factor > 0.9 in magnitude and eliminated redundant features, using guidance from Global and per-trace Pearson's factor of those features. By doing this, I reduced the feature count to 9. Thus, in the final implementation of PPF, no two features have a high correlation between them. This way we can be sure that each feature makes a contribution that cannot be captured using other features.

Secondly, studying the relative importance of each feature enabled me to vary the number of entries dedicated for each feature. Features with higher correlation, *cache line* and *page address* were given most importance and allowed full 12-bits of indexing. Features like *PC XOR delta* and *PC XOR depth* with a low overall P-value were allocated fewer entries in the feature table.

Field	Bits	Comment
Valid	1	Indicates a valid entry in the table
Tag	6	Identifier for the entry in the table
Useful	1	To show if the given entry led to a useful demand fetch
Perc Decision	1	Prefetched vs Not-prefetched
PC	12	Metadata required for perceptron trainng
Address	24	
Curr Signature	10	
PC_i Hash	12	
Delta	7	
Confidence	7	
Depth	4	
Total 85 bits		

Table 4.2: Metadata Stored in Prefetch Table

Structure	Entry	Components	Total
Signature Table	256	Valid (1 bit) Tag (16 bits) Last Offset (6 bits) Signature (12 bits) LRU (6 bits)	11008 bits
Pattern Table	512	C_{sig} (4bits) C_{delta} (4*4 bits) Delta (4*7 bits)	24576 bits
Perceptron Weights	4096*4 2048*2 1024*2 128*1	5 bits	113280 bits
Prefetch Table[1]	1024	85 bits	87040 bits
Reject Table[2]	1024	84 bits	86016 bits
Global History Register	8	Signature (12 bits) Confidence (8 bits) Last Offset (6 bits) Delta (7 bits)	264 bits
Accuracy Counters	1 1	C_{total} C_{useful}	10 bits 10 bits
Global PC Trackers	3	PC_1 (12 bits) PC_2 (12 bits) PC_3 (12 bits)	36 bits
Total: 3,222,940 bits = 39.34 KB			

Table 4.3: SPP-Perc Storage Overhead

4.1.6 Overhead for PPF

In this section, I analyze the hardware overhead required to implement PPF. The Prefetch Table was enhanced to accommodate storing of metadata for perceptron training. Table 4.2 depicts the metadata stored for each entry in the Prefetch Table. Table 4.3 shows the total storage overhead of PPF implementation. The hardware budget for 2nd Data Prefetching championship was 32 KB. Keeping that in mind the considerable speedup PPF obtained over the winner, the extra hardware budget can be accounted for. The extra hardware also makes the overall scheme more scalable than SPP. In the original SPP paper, it was demonstrated that adding extra hardware brings little

advantage in terms of performance gain. The newly added perceptron tables can be scaled to increase / decrease features depending on the permitted budget.

In terms of computations, the perceptron mechanism only introduces an extra adder tree. The hash perceptron mechanism makes sure than there is no actual vector multiplication happening in the hardware. Obtaining the perceptron sum requires addition of nine 5-bit numbers. Using an adder tree of four 5-bit adders, this can be done in ceil($log_2 9$) = 4 steps. Perceptron update only requires weight update by +1 or -1. Thus, all the operations required for perceptron inferencing or updating the states of the perceptrons can be easily done in the time constraints of L2 Cache Accesses.

[1]Components of Prefetch Table can be found in Table 4.2.

[2]The Reject Table does not need to maintain the useful bit as that only applies for prefetches that ultimately made through.

4.2 Results

This section discusses the results obtained from running PPF in terms of speedup and prefetch cache, for the SPEC CPU 2017 benchmarks. First, I present the results for single-threaded workloads then for multi-core workloads.

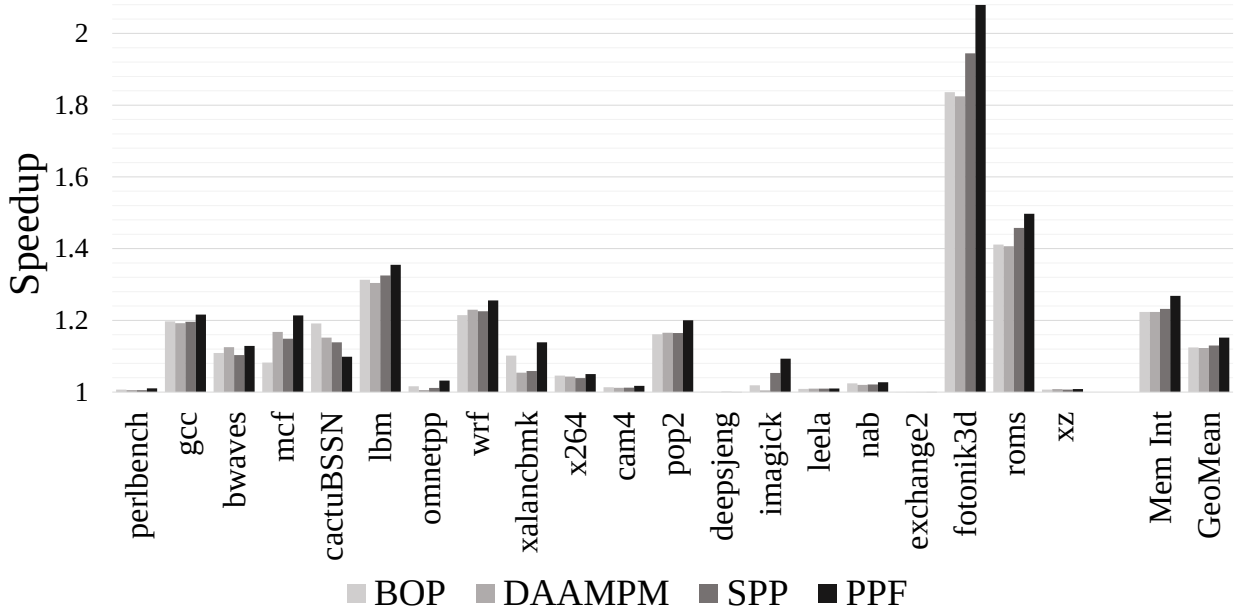


Figure 4.4: SPEC CPU 2017 Single-Core IPC Speedup

4.2.1 Single-core Results

Figure 4.4 shows the single core speedup obtained by BOP, DA-AMPM, SPP and PPF for each of the individual SPEC CPU 2017 applications, followed by the geomean of the memory intensive subset and finally the geomean across the full suite. All the results have been normalized to the baseline of no prefetching.

PPF yields a geometric mean speedup of **26.95%** over the baseline. This is equivalent to **4.63%** over DA-AMPM, **4.61%** over BOP and **3.78%** over SPP. Out of the 20 SPEC CPU 2017 applications, PPF nearly matches or outperforms all the other prefetchers on 19 applications.

Benchmarks `603.bwaves_s`, `605.mcf_s`, `623.xalancbmk_s` and `649.fotonik3d_s` benefit the most from PPF, with the speedup over SPP ranging from **10% to 25%**.

One interesting case here is `623.xalancbmk`. Despite SPP under performing on that application, PPF manages to considerably outperform all prefetchers. Since this benchmark has varying prefetch deltas, SPP's conservative throttling mechanism catches that and quickly halts prefetching at an average depth of 2.1. On the other hand, PPF's more efficient accuracy check enables it to prefetch upto a lookahead depth of 3.3. Doing this, PPF suggests 1.61 times more total prefetches and 2.53 times more useful prefetches than SPP.

The only benchmark where PPF fails to match the improvement offered by other prefetchers is `607.cactuBSSN_s`. Based on the observations of prefetching behavior, I gather that BOP's aggressive and localized nature fits this workload very well; as opposed to SPPs lookahead nature. As a result, SPP, and hence PPF, underperform on this benchmark.

On the full SPEC CPU 2017 suite, PPF improves the geometric mean IPC of the baseline by **15.24%**, which is **2.27%** better than the next best prefetcher – SPP.

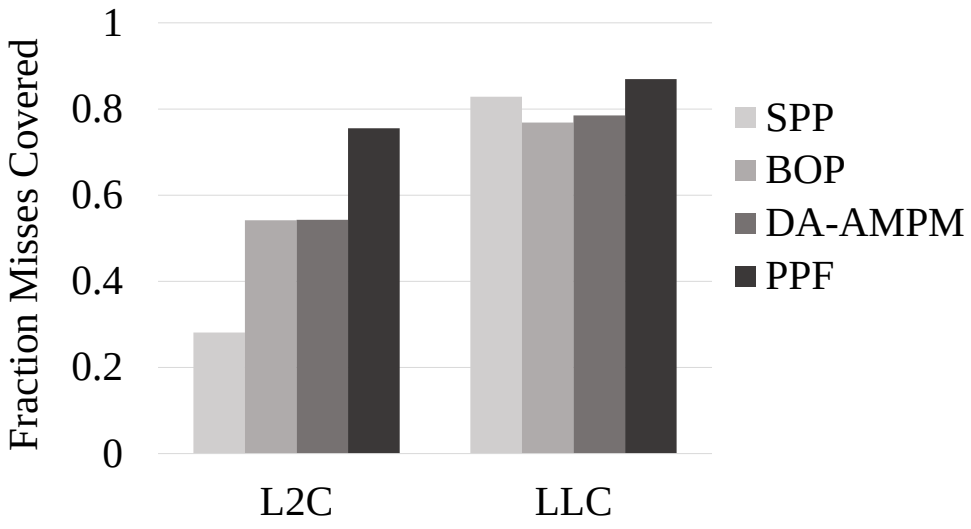


Figure 4.5: Fraction of Cache Misses Covered

Coverage: Prefetcher coverage is defined as the ratio of the number of misses avoided through prefetching over the number of misses with no prefetching. Figure 4.5 shows the fraction of misses in the L2 and LLC avoided by the various prefetchers. PPF has the highest coverage of all the prefetchers simulated. On the SPEC CPU 2017 benchmarks, PPF reduces misses by **75.5%** and **86.9%** in the L2 and LLC respectively. For the same benchmarks, the next best prefetcher, DA-AMPM, covers **54.3%** and **78.5%** of the misses respectively.

This superior coverage of PPF can be attributed to aggressive re-tuning of the underlying SPP, enabled by the Perceptron Filter making sure the high coverage does not lead to increased cache pollution.

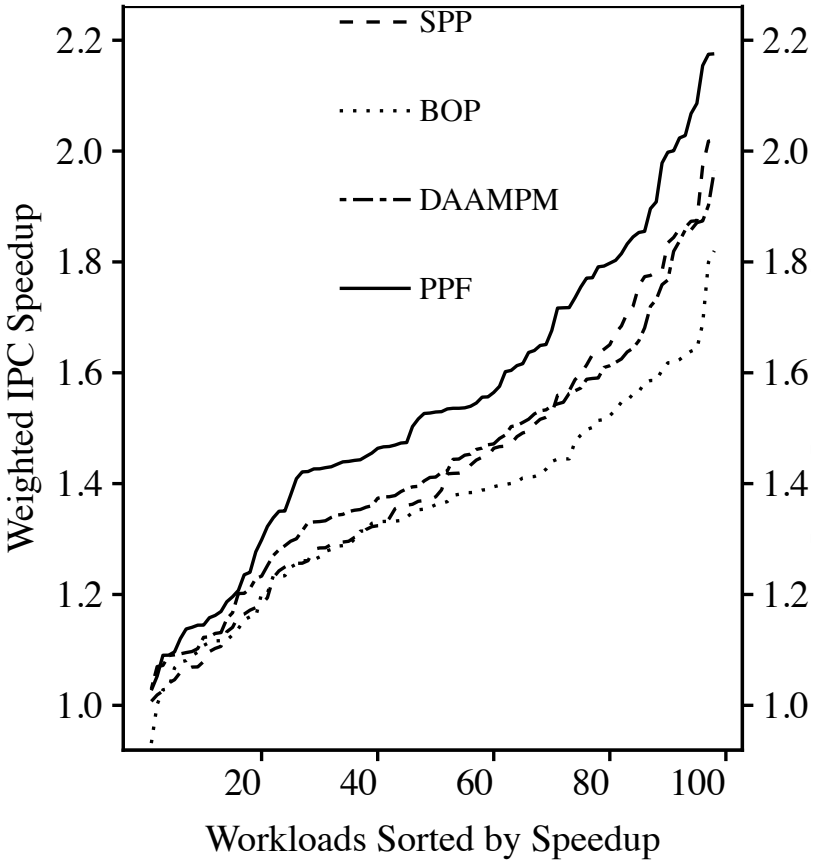


Figure 4.6: Speedup for 4-core SPEC CPU 2017

4.2.2 Multi-core Results

In this section, I demonstrate the improvement achieved by PPF for a mix of multi-programmed workloads.

4-Core: Figure 4.6 shows a comparison of speedups obtained on 4-core mixes of a memory intensive subset of SPEC CPU 2017. I plot all 4 prefetchers, normalized to the baseline. The workloads have been sorted in increasing order of the speedup. PPF offers a speedup of **51.2%** on these traces, an improvement of **11.4%** over the underlying SPP, **9.7%** over the next DA-AMPM, and **16.9%** over BOP. On a different set of fully random SPEC CPU 2017 4-core mixes (not illustrated for space reasons), PPF provides an IPC speedup of **26.07**% over the baseline, which is an improvement of **5.6**% over SPP.

8-Core: The sorted comparison of speedups on the memory intensive 8-core mixes is shown in Figure 4.7. PPF improves baseline performance by **37.6%**, an improvement of **9.65%** over SPP. For a random set of SPEC CPU 2017 mixes (not illustrated for space reasons), PPF improves performance by **23.4%** over the baseline, corresponding to **4.6%** over SPP. This increased improvement achieved by PPF over the base engine SPP in a multi-core environment is expected as PPF is a very accurate filter. Thus, it eliminates useless prefetches before they can cause pollution in the shared LLC. BOP offers a better improvement than SPP for the memory intensive mixes. This superiority can be attributed to BOP's inherent aggressive nature. DA-AMPM is also ahead of SPP in both the mixes. Interestingly, in all these cases, PPF consistently outperforms the best performing prefetcher.

4.2.3 Additional Memory Constraints

I also model PPF with reduced LLC and with low bandwidth constraints, respectively (not illustrated for space reasons). Benchmark `605.mcf_s` in low bandwidth conditions is prefetch averse. In general, any prefetcher yields a negative speedup on that trace. On `654.roms_s` and `607.cactuBSSN_s`, PPF is unable to match the performance achieved by the best prefetcher. On the other hand, PPF outperforms all the other prefetchers on `623.xalancbmk_s` and `638.imagick_s` benchmarks. Overall, PPF provides a greater improvement under small LLC

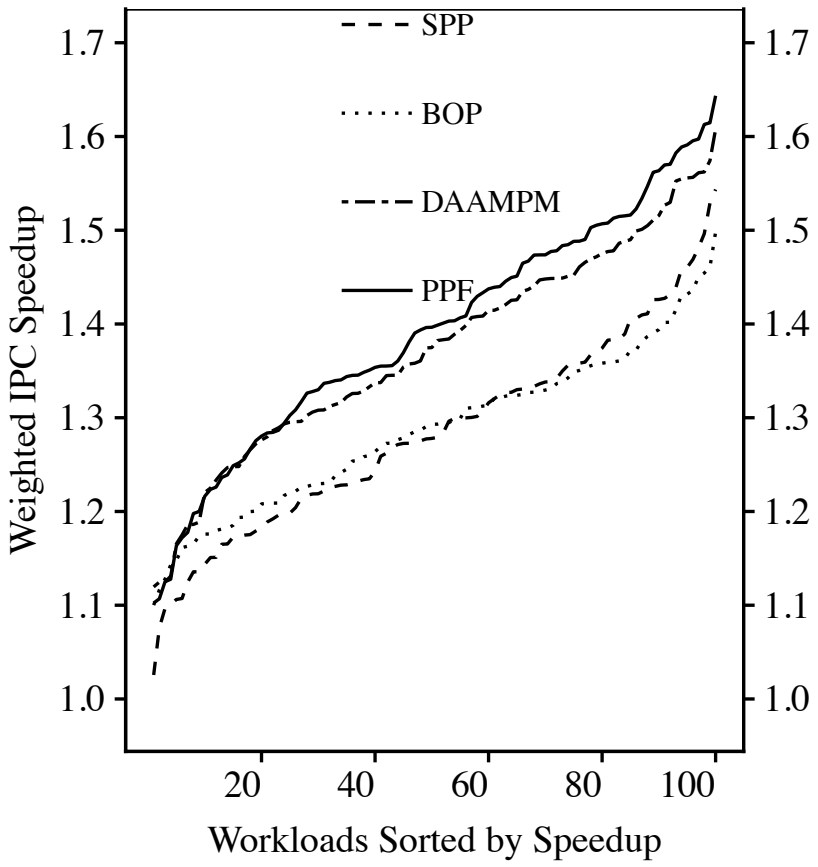


Figure 4.7: Speedup for 8-core SPEC CPU 2017

condition and matches the best prefetcher, BOP, under low DRAM bandwidth conditions.

4.2.4 Cross Validation

Figure 4.8(a) shows the performance benefit comparison of all the prefetch schemes on 4 different applications in the CloudSuite benchmark. In general, these applications are prefetch agnostic. Even so, PPF manages a **3.78%** improvement over no prefetching, putting it ahead of the next best prefetcher, SPP, which provides a 3.08% speedup.

Figure 4.8(b) shows the speed-up achieved on the memory intensive subset and the full SPEC CPU 2006 suite for a single-processor machine. PPF provides a speedup of **36.3%** over the baseline on the memory intensive subset of SPEC CPU 2006 benchmark, giving an improvement of **6.1%**

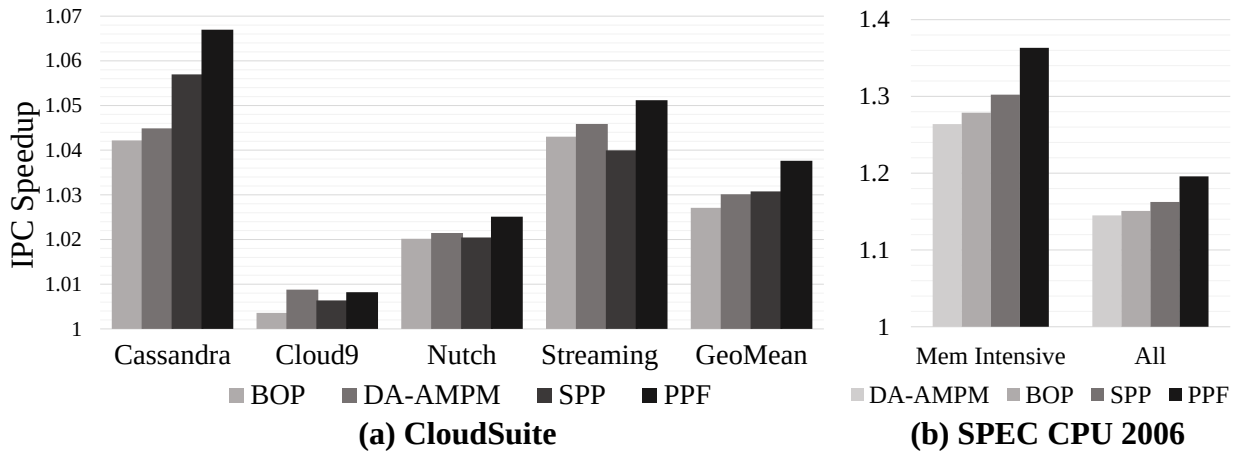


Figure 4.8: IPC Speedup for Unseen Workloads

over SPP and **8.44%** over DA-AMPM and **9.93%** over BOP. On the whole of the SPEC CPU 2006 suite, the speedup is **19.6%**, an improvement of **3.33%** over SPP.

For 4-core memory intensive mixes, PPF improves the baseline by **59.1%**, **8.6%** ahead of SPP. For 8-core memory intensive mixes, the speedup over the baseline is **47.8%**, **11.3%** ahead of SPP.

I developed PPF to yield good performance on the SPEC CPU 2017 benchmarks. Nevertheless, the performance is consistently good on other benchmark suites. I attribute this fact to the inherent adaptability of the perceptron model. In general, perceptron weights are able to adjust in real-time so as to find the best possible correlation between the output and the given set of features.

5. CONCLUSION

5.1 Conclusion

In this thesis, I introduce Perceptron-Based Prefetch Filtering (PPF). PPF is an on-line perceptron learning based classifier. It classifies the prefetch suggestions coming from the underlying prefetch engine as useless prefetches vs useful prefetches. Furthermore, it classifies the useful prefetches as high-confidence vs low-confidence useful, to decide the fill level of the prefetches. Thus, PPF acts as an independent check on the quality of predictions made by the baseline prefetcher. This two-step approach helps overcome the inherent trade-off between the aggressiveness and the accuracy of a prefetcher and increase performance of the overall prefetchig scheme. I also created a case study implementation of PPF using SPP as the baseline prefetch engine, while in principle other prefetchers could be used.

Some of the merits of using PPF based approach are:

- **Performance Boost:** By effectively filtering out bad prefetches, PPF helps increase the overall performace (measured in terms of IPC) of the system by up to 11.4%, as compared to just using the baseline prefetcher. This makes PPF perform even better on constrained environments like a multi-core system.

- **Quick On-line Learning:** The huge advantage of using the perceptron-based learning approach is that it can easily learn the memory access patterns of any application and even quickly adapt to dynamically changing program behaviour. This makes the prefetching scheme highly robust and versatile.

- **Modular Approach:** A large part of the perceptron learning remains hidden from the underlying prefetcher. PPF can be as loosely or as tightly coupled with the baseline prefetcher, depending on the availability of the metadata to be conveyed between the two. This makes PPF an attractive add-on to any future prefetching schemes.

- **Scalable Design:** The feature-set of PPF can be scaled up / down, depending on the available hardware budget. Controlling the bit-width of each perceptron and independent depth of each perceptron table gives further control over achieving the performance vs hardware trade-off for PPF.

I show that PPF effectively filters bad prefetches, so that the given underlying prefetcher can be highly aggressively tuned to achieve increasing coverage. PPF is a robust and adaptable technique that can be used to enhance any existing prefetcher and can be a valuable tool in the design of future memory latency constrained systems.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[2] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[3] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: understanding the effects of multiprocessor prefetches," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 177–188, March 2006.

[4] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 285–297, June 2015.

[5] S. Liao, T. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine learning-based prefetch optimization for data center applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–10, Nov 2009.

[6] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), pp. 364–373, ACM, 1990.

[7] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec 1978.

[8] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, Dec. 1978.

[9] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, (New York, NY, USA), pp. 176–186, ACM, 1991.

[10] J.-L. Baer and T.-F. Chen, "Effective hardware-based data prefetching for high-performance processors," *IEEE Trans. Comput.*, vol. 44, pp. 609–623, May 1995.

[11] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Making address-correlated prefetching practical," *IEEE Micro*, vol. 30, pp. 50–59, Jan 2010.

[12] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 499–500, ACM, 2009.

[13] C. F. Chen, S. . Yang, B. Falsafi, and A. Moshovos, "Accurate and complexity-effective spatial pattern prediction," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pp. 276–287, Feb 2004.

[14] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, (Washington, DC, USA), pp. 252–263, IEEE Computer Society, 2006.

[15] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 1–10, IEEE Computer Society, 2008.

[16] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pp. 79–90, Feb 2009.

[17] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 69–80, ACM, 2009.

[18] S. Somogyi, T. F. Wenisch, M. Ferdman, and B. Falsafi, "Spatial memory streaming," *J. Instruction-Level Parallelism*, vol. 13, 2011.

[19] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jiménez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 623–634, Dec 2014.

[20] L. M. AlBarakat, P. V. Gratz, and D. A. Jiménez, "Mtb-fetch: Multithreading aware hardware prefetching for chip multiprocessors," *IEEE Computer Architecture Letters*, vol. 17, pp. 175–178, July 2018.

[21] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: Memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (NY, USA), pp. 267–278, ACM, 2012.

[22] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. L. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 626–637, Feb 2014.

[23] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, March 2016.

[24] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 141–152, Dec 2015.

[25] 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," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, (New York, NY, USA), pp. 737–749, ACM, 2017.

[26] C. Wu, A. Jaleel, M. Martonosi, S. C. Steely, and J. Emer, "Pacman: Prefetch-aware cache management for high performance caching," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 442–453, Dec 2011.

[27] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, pp. 51:1–51:22, Jan. 2015.

[28] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 355–366, ACM, 2012.

[29] A. Jain and C. Lin, "Rethinking belady's algorithm to accommodate prefetching," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 110–123, June 2018.

[30] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, pp. 197–206, 2001.

[31] D. Tarjan and K. Skadron, "Merging path and gshare indexing in perceptron branch prediction," *ACM Trans. Archit. Code Optim.*, vol. 2, pp. 280–300, Sept. 2005.

[32] E. Teran, Z. Wang, and D. A. Jiménez, "Perceptron learning for reuse prediction," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 2:1–2:12, IEEE Press, 2016.

[33] D. A. Jiménez and E. Teran, "Multiperspective reuse prediction," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), pp. 436–448, ACM, 2017.

[34] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on*

Microarchitecture, MICRO '43, (Washington, DC, USA), pp. 175–186, IEEE Computer Society, 2010.

[35] J. A. Joao, O. Mutlu, C. J. Lee, R. Cohn, Y. N. Patt, and H. Kim, "Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware," *IEEE Transactions on Computers*, vol. 58, pp. 1153–1170, 12 2008.

[36] "The champsim simulator,"

[37] S. H. Pugsley, A. R. Alameldeen, C. Wilkerson, and H. Kim, "The 2nd data prefetching championship (dpc-2),"

[38] "The 2nd cache replacement championship (crc-2),"

[39] "Standard performance evaluation corporation cpu2017 benchmark suite,"

[40] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, (New York, NY, USA), pp. 318–319, ACM, 2003.

[41] "Standard performance evaluation corporation cpu2006 benchmark suite,"

[42] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 37–48, ACM, 2012.