Deep-Learning based Filter for Hardware Prefetchers

Submitted in partial fulfillment of the requirements for the degree of
Master of Technology
by

Debasish Das
203050103
debasish@cse.iitb.ac.in

Supervisor:
Biswabandan Panda

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
2020-2022

Approval Sheet

This report entitled "Deep-Learning based Filter for Hardware Prefetchers" by Debasish Das is approved for the degree of Master of Technology in Computer Science and Engineering.

Examiners

Digital Signature
Shivaram Kalyanakrishnan (i15034)
28-Jun-22 04:14:51 PM

Digital Signature
Preethi Jyothi (i16268)
27-Jun-22 07:14:17 AM

Digital Signature
Biswabandan Panda (10001949)
29-Jun-22 12:16:39 AM

Supervisor

Chairman

Digital Signature
Preethi Jyothi (i16268)
29-Jun-22 10:43:00 AM

Date: 27th June, 2022
Place: Indian Institute of Technology Bombay

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Digital Signature
Debasish Das (203050103)
29-Jun-22 04:07:51 PM

Debasish Das
203050103

Date: 29th June, 2022

Abstract

Prefetching is one of the techniques that is widely used to *hide* the memory latency by bringing the data into the caches before the processor asks for it. Thus, for a prefetcher to be effective, it must not only be *accurate* in its prefetching decisions, but also be *timely* because transferring data from the main-memory to the caches takes time. Even if a prefetcher is 100% accurate, if it fails to achieve timeliness, then the utility of prefetching reduces and it might reduce the performance of a system instead because of the increased memory traffic. Conversely, even if the prefetcher is 100% timely, but is not accurate, it will drastically reduce the performance of a system, especially under low memory bandwidth due to increased miss penalty and eviction of possibly useful cache-lines. This work proposes a deep-learning based filter that can be *augmented* with *any* prefetcher and can improve timeliness as well as accuracy of the prefetchers. Although it is a proof-of-concept, it shows promising results by achieving near-similar performance by significantly reducing the memory traffic.

Contents

Approval Sheet i

Declaration ii

Abstract iii

1 Introduction 1

2 Reinforced Signature-Path Prefetcher 3
- 2.1 Motivation . . . 3
- 2.2 Prefetching as a Reinforcement Learning Problem . . . 5
- 2.3 High-Level Overview . . . 7
  - 2.3.1 Issuing Rewards . . . 7
  - 2.3.2 Issuing Prefetches . . . 9
- 2.4 Detailed Overview . . . 9
  - 2.4.1 Tracking Table . . . 9
  - 2.4.2 Extended Pattern Table . . . 11
- 2.5 Evaluation . . . 11
  - 2.5.1 Simulation Infrastructure and Benchmarks . . . 11
  - 2.5.2 Results . . . 11
  - 2.5.3 Prefetcher Configurations . . . 12

3 Neural-Filter 15
- 3.1 Motivation . . . 15
- 3.2 Model Architecture . . . 16
  - 3.2.1 Byte-Distributed Embedding with Attention . . . 17
  - 3.2.2 Variable-Width Convolutions . . . 22
  - 3.2.3 Bi-GRU with Attention . . . 23
  - 3.2.4 Prediction Feed-Forward Networks . . . 24
- 3.3 Model Configuration and Dataset . . . 25
  - 3.3.1 Model Configuration . . . 25
  - 3.3.2 Dataset for Training, Validation and Testing . . . 26

4 Results **29**
4.1 Misclassification Rate on Test Data 29
4.2 Late Prefetches Issued 30
4.3 IPC Improvement 34
4.4 Reduction in Accesses to Off-Chip Memory 34
4.5 Prefetch Accuracy 35
4.6 Summary 37

5 Conclusion **41**

6 Acknowledgements **43**

Chapter 1

Introduction

Memory-Wall is the well-known performance gap between the processors and the main-memory, i.e. DRAMs, which has gotten worse over the years due to the significant amount of research put into processors to make them execute more number of instructions per clock-cycle as compared to the amount of research put into improving the transfer rates of DRAMs. This resulted in systems being bottle-necked by the performance of main-memory. Tons of approaches to overcome this problem have been proposed over the years, one of which is the well known multi-level memory hierarchy, i.e. the introduction of caches between the processor and the main-memory. Although this has reduced the gap by a significant amount, there is still a limit to the amount by which the caches can be scaled up, while keeping in mind the power-consumption and the cost-per-bit associated with them.

Prefetching is one technique that tries to reduce the performance impact caused by long transfer times between main-memory and caches, by bringing the data into the caches *before* the processor actually requires it. Thus a successful prefetch effectively *hides* the entire latency associated with the data transfer. An ideal prefetcher gives the illusion that the caches are of unlimited capacity. Because nothing is ideal in this world, prefetching has its own challenges and its trade-offs. More specifically, the challenges are *what*/*when*/*where*/*how* to prefetch, each of which, has its own set of trade-offs.

This work begins by focusing on the *when* to prefetch aspect by analyzing one prior (hardware-based) state-of-the-art (SOTA) prefetcher for level-2 caches (L2C), *Signature Path Prefetcher* (SPP) [11], which focuses only on *what* and *where* to prefetch aspects [1] without considering the *when* to prefetch aspect. It then proposes an improvement to SPP's design by also incorporating the *when* to prefetch aspect into the prefetcher's design, with an objective to further improve the performance of the prefetcher.

Second part of the work analyzes the drawbacks of the approach used in improving the performance of SPP as well as the drawbacks of current prefetchers under low memory bandwidth and then proposes a general-purpose framework that can be *augmented* with *any* hardware-prefetcher. It does so by modifying the previous objective of improving only

[1] The *how* to prefetch aspect is fixed at design time, for ex. hardware-based/software-based prefetching etc.

timeliness and also including another objective of minimizing the amount of memory traffic by limiting the number of prefetch requests while minimizing the impact it causes on the overall performance.

Chapter(2) describes the task of prefetching in detail and the enhancement proposed for SPP, i.e. incorporating timeliness, via policy-based reinforcement learning, resulting in *Reinforced SPP* (R-SPP) - the proposed timely prefetcher, which is the focus of first part of this work. It then evaluates the proposed *proof-of-concept* design on SPEC2017 benchmarks. Chapter(3) builds upon the observations from chapter(2) as well analyzing the drawbacks of two more SOTA prefetchers - IPCP [14] and Bingo [2]. It then modifies the objective of improving only timeliness by also including another objective of minimizing the amount of memory traffic while minimizing the overall negative performance impact it causes. It proposes another proof-of-concept model based on *supervised deep-learning*, *Neural-Filter* (NeuFi), to achieve the previously mentioned objective. Chapter(4) evaluates the proposal of chapter(3) by conducting in-depth experiments by augmenting NeuFi with IPCP [14], Bingo [2] and SPP [11], on SPEC2017 and GAP benchmarks. Finally, chapter(5) concludes this work by summarizing the results obtained.

Chapter 2

Reinforced Signature-Path Prefetcher

2.1 Motivation

Prefetching is one of many techniques that is employed to overcome the *memory-wall* problem, i.e. the performance gap between the processor and main memory, which has grown significantly over the years. The objective of prefetching is to *hide* the expensive memory latency that is incurred on transferring the data from the main-memory into the caches. It is achieved by bringing the data into the caches, before the processor actually needs them. One of the core challenges in designing an effective prefetcher is to make the prefetcher learn *when* to initiate a prefetch request, apart from *what* to prefetch, *where* to prefetch and *how* to prefetch. Given a prefetcher, even if it knows the exact answers to what and where to prefetch [1] on every cache access, if timeliness is not considered, then the prefetches that were issued, will fall into one of the following three classes

1. **Late Prefetches** - When a cache-line is prefetched and the demand request for it arrives *before* the prefetch is completed. The prefetch holds no meaning in this case, if the demand request arrives immediately after the request was issued because it could not even hide a fraction of the latency

2. **Timely Prefetches** - When a cache-line is prefetched and the demand request for it arrives *not long after* the prefetch is completed. This is the ideal case for any prefetcher

3. **Early Prefetches** - When a cache-line is prefetched and the demand request for it arrives *too late* (or never), *after* the prefetch is completed. The prefetch holds no meaning if, whatever it brought, never gets used. This might evict useful data from the cache, thus degrading the performance instead

Signature Path Prefetcher (SPP) [11] is a prior state-of-the-art prefetcher, which is although able to accurately and quickly learn complex memory access patterns, suffers from one key deficiency which makes the prefetcher unable to utilize its full potential - *timeliness*

[1] The *how to prefetch* affects the design of a prefetcher, for ex. hardware-based/software-based etc.

of the prefetches. As proven by the experiments [2] shown in fig(2.1) and fig(2.2), the degree of late prefetches increase when the available DRAM bandwidth per-core decreases. The problem lies in the fact that the prefetches are issued without considering the timeliness of the prefetching decisions.

As will be described in section(2.2), *most of the traditional prefetchers are inherently weak reinforcement-learning (RL) agents*, this work tries to incorporate timeliness into SPP's prefetching algorithm via the RL-framework while ensuring that the algorithm is able to adapt to different workloads and system configurations.

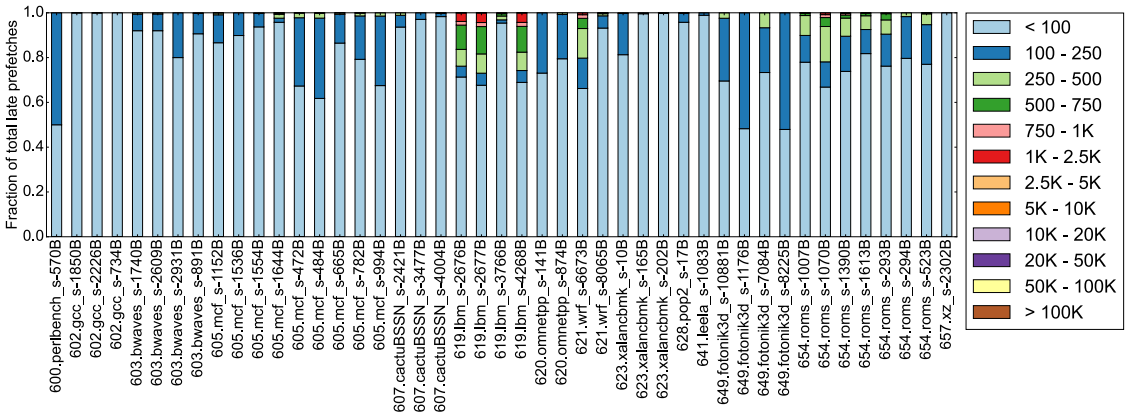


Figure 2.1: Distribution of late prefetches issued by SPP [11] on a single-core system with single-channel DRAM with bandwidth of 3200 MT/s (the default value), simulated using ChampSim. The legend on the right-side indicates the number of CPU cycles by which the prefetches were late.

It is a well known fact that the reinforcement-learning algorithms are very *sample inefficient* and are notoriously *slow* in learning successful policies, especially in highly stochastic environments. Thus these algorithms lack the ability to rapidly latch onto successful strategies[5]. In a resource constrained and rapidly-changing environment, where rapidly adapting the policy without incurring extreme performance/storage overhead is desired, learning the state-action *Q-values* by the prefetcher does not seem to be good fit because, as mentioned previously, getting an accurate estimate on them requires significant amount of data. Even if they are successfully learnt, context-switches to different processes with different access patterns and behavior happen very frequently in modern-day systems. So these *learnt* state-action values lose their utility because the dynamics of the environment change when there is a context switch and hence, there is a need to re-learn these values all over again, which again will take time.

Keeping these things in mind, this work instead adopts a different form of policy update, inspired from [5]. Instead of maintaining *Q-values* of each possible action for each possible

[2]Credits to Sumon Nath, a first-year MS student @IITB, for modifying ChampSim to gather the distribution of late prefetches by CPU cycles and allowing me to use his work for experimenting with SPP

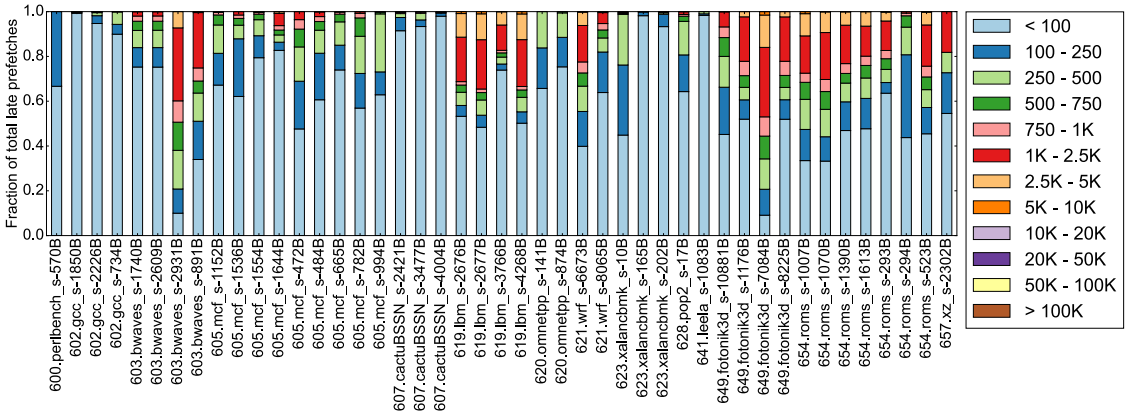


Figure 2.2: Distribution of late prefetches issued by SPP [11] on a single-core system with single-channel DRAM with bandwidth of 400 MT/s, simulated using ChampSim (with all other parameters kept constant as before). The legend on the right-side indicates the number of CPU cycles by which the prefetches were late.

state, it instead maintains *integer* values quantifying how *good* or *bad* each decision is, for each possible state [3]. Rapidly modifying the policy automatically to adapt to a completely different environment is possible here because rewarding the action simply means incrementing/decrementing the value by some fixed amount, as explained in the later sections.

2.2 Prefetching as a Reinforcement Learning Problem

Suppose (P_k, b_k, h_k) denote a pair of random variables for k'th access ($k > 0$) to the cache, for page P_k and cache-line b_k within the page, with $h_k \in \{0, 1\}$ denoting whether it was a hit ($h_k = 1$) in the cache or not. Let a sequence of demand requests, a.k.a. demand request stream, to the cache be denoted as D_{stream}, i.e.

$$D_{stream} = (P_1, b_1, h_1), (P_2, b_2, h_2), \cdots \tag{2.1}$$

The demand request stream can be modelled as a directed acyclic graph as follows where the directed edges represent the transition between successive accesses. Since the next access can either be a hit or a miss (and similar thing for the current access), both in-degree and out-degree of each state (except the first one) is two.

Given any access (P_k, b_k, h_k), the objective of a prefetcher is to ensure that *majority* (if not all) of the subsequent accesses $(P_{k'}, b_{k'}, h_{k'})$, $\forall k' > k$, end up as a hit, i.e. $h_{k'} = 1$ (which would have otherwise missed without the prefetcher), so as to *maximize* the throughput of a system over the long run. In an ideal scenario, the prefetcher would ensure that transitions

[3]For now every (state, action) pair is stored. Optimizing the storage, similar to [5] is a work left for future

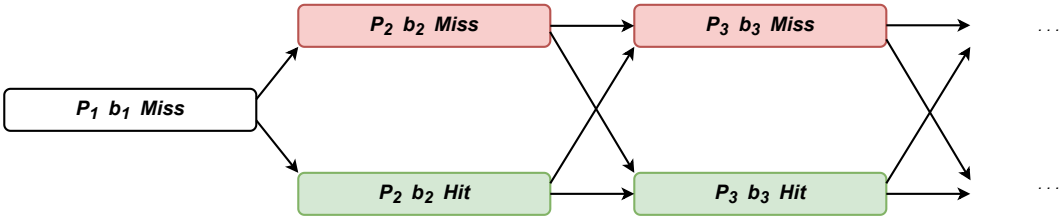


Figure 2.3: Demand request stream modelled as a directed acyclic graph

in the access transition graph from fig(2.3) only happen from states with $h_k = 1$, i.e. any state for which $h_k = 0$ is never visited, apart from the first access which will be always be a miss, during the entire demand request stream.

What many prefetchers (for ex. [2], [11], [13] [14], etc.) essentially do is that they try to model the behavior of D_{stream}, based on the observations seen so far and sends the prefetches accordingly. Because D_{stream} is a stochastic process, the estimated model ends up being probabilistic in nature. Mathematically, given that current state in fig(2.3) is (P_k, b_k, h_k), the prefetchers estimate some form of [4]

$$\mathbb{P}\{P_{k+1}, b_{k+1} \mid P_1, b_1, P_2, b_2, \cdots, P_k, b_k\}, \quad \forall k > 0 \tag{2.2}$$

and sends $n(\geq 0)$ prefetch requests to the n highest probable (P_i, b_i) pairs, $\forall i > k$, based on the estimates obtained from eq(2.2). Because practically, the computation of eq(2.2) becomes intractable quickly, some simplifying assumptions are made. To be more specific, an i'th order $(i > 0)$ *Markov* assumption is made, i.e.

$$\mathbb{P}\{P_{k+1}, b_{k+1} \mid P_1, b_1, P_2, b_2, \cdots, P_k, b_k\} \approx \mathbb{P}\{P_{k+1}, b_{k+1} \mid P_{k-i+1}, b_{k-i+1}, \cdots, P_k, b_k\} \tag{2.3}$$

Eq(2.3) basically states that the next state only depends on the most-recent i states, not on any state prior to them.

The model D_{stream} can be thought of as a dynamically changing *environment* over time, with *good states* being the ones for which $h_k = 1$. The prefetchers are the *agents* that are acting on this environment whose *actions* can be thought of as the prefetching decisions they make on each (P_k, b_k, h_k) so as to drive the future states towards having $h_{k'} = 1$, $\forall k' > k$, which would lead to obtaining the highest throughput from the system. The *feedback* given by the environment (i.e. *rewards*) for each such decision to (or not to) prefetch (P_i, b_i) arrive after $x_i(> 0)$ accesses (i.e. delayed reward scheme) in the form of a hit or a miss. There are two possibilities for each such decision

1. **Prefetch was issued for** (P_i, b_i)
   (a) Demand request for it arrived on $(k + x_i)$'th access and it resulted in a *hit* due to prefetch

[4] $h_{\cdot}$ is irrelevant for the prefetcher during model estimation because this is what it tries to influence

(b) Demand request for it *never* arrived ($x_i = \infty$)

2. **Prefetch was not issued for** (P_i, b_i)

   (a) Demand request for it arrived on $(k + x_i)$'th access and it resulted in a *miss* due to not prefetching it

   (b) Demand request for it *never* arrived ($x_i = \infty$)

It is immediately apparent from the above on what constitutes a good action and what constitutes a bad action. Because it is not possible to know in advance on what an optimal set of actions for each (P_k, b_k, h_k) would be (as it is not possible to know the future), the prefetcher must learn to find them either through exploration or just simply bootstrap with whatever estimate of the model, the prefetcher currently has. So, in a nutshell,

Most of the traditional prefetchers are inherently weak reinforcement learning agents

What makes them *weak* is because the prefetching *policy* is *never* updated [5] for bad decisions made in the past (atleast for the bad prefetches that were issued), but only the estimate of the model is refined over time. This is analogous to ignoring the mistakes made in the past and not trying to rectify them, so as to avoid doing the same thing in the future. Hence, if the prefetching policy is also appropriately updated over time along with the model estimate, this would lead to the design of more optimal prefetchers that can adapt to the dynamically changing environment and as a result, make optimal prefetching decisions that can improve the throughput of a system significantly.

2.3 High-Level Overview

Figure(2.4) provides a high-level overview on the working of *Reinforced Signature Path Prefetcher* (R-SPP). Similar to *Signature Path Prefetcher* (SPP) [11], the main modules are almost the same, except for few modifications and one new addition: *Tracking Table* (TT) [6]. The main objective of TT is to track the prefetching decisions that were issued in the past for some amount of time and then use it update the prefetching policy of R-SPP accordingly - encouraging good decisions and penalizing bad decisions. Tracking the prefetching decisions are required because of the delayed reward scheme, as described previously in section 2.2, so that bad decisions can be penalized and vice-versa for good decisions.

2.3.1 Issuing Rewards

When a demand request for a cache-line (or block) b_i in page P_i arrives to level-2 cache (L2C), the TT is searched for a past prefetching decision that might have prefetched (P_i, b_i), similar to [4]. If an entry is found, then it is checked whether that prefetch was a timely prefetch or not

[5] One exception is *Berti* [15] prefetcher that updates the policy for prefetches not issued

[6] This is almost similar to *Evaluation Queue* (EQ) of Pythia [4]

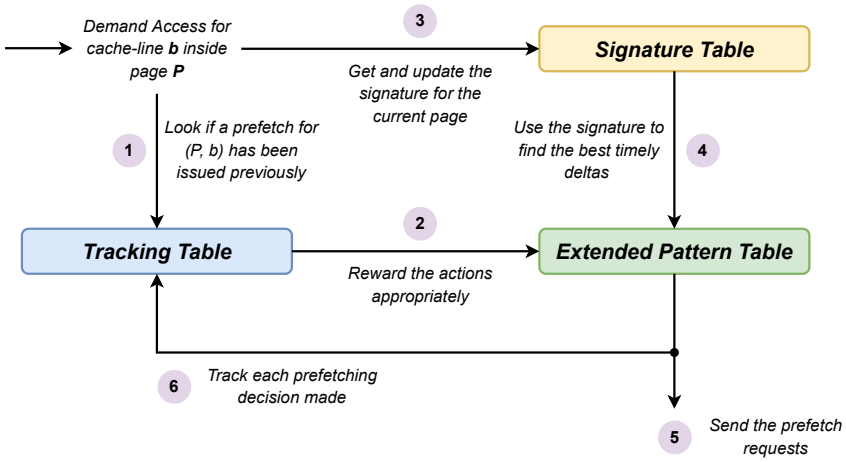


Figure 2.4: High-level overview of how R-SPP operates.

1. **Prefetch was timely**: A *positive* reward is given for the prefetching decision, by incrementing the confidence of the action, i.e. the delta (defined below in eq(2.4)) associated with its corresponding signature, i.e. the delta associated with the signature that was used to prefetch (P_i, b_i) is given a positive reward.

2. **Prefetch was not timely**: A *negative* reward is given for the prefetching decision, by decrementing the confidence of the action associated with its corresponding signature, similar to what was done above. The main objective is to discourage the prefetcher from sending such prefetches in the future.

Define *delta* (d_k) on k'th access $(k > 1)$ to be the difference between cache-line number accessed on $k'th$ access and on $(k-1)$'th access

$$d_k = \begin{cases} b_k - b_{k-1} & \text{if } k > 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

Irrespective of whether or not a matching entry was found in the TT, all other entries e_j, that are currently awaiting for their rewards, i.e. the *most-recent* prefetches, are checked if their prefetches also belong to the same page P_i and if they do, they are checked if they could also have also brought b_i in a timely manner (inspired from [15]). If yes, then the action corresponding to e_j's signature, that could have brought b_i is also given a positive reward. The point is to encourage others to also prefetch b_i in a timely manner, so as to break the strict dependence that only a single signature stores the information for fetching b_i in a timely manner. This step is one of the crucial steps in making the prefetcher *learn* what it should have done instead, thus providing some form of a supervision update.

2.3.2 Issuing Prefetches

After rewards are issued, the signature for the corresponding demand request is extracted, similar to what is done in SPP [11] and then it is used to query the *Extended Pattern Table* (EPT), i.e. an extended version of SPP's *Pattern Table*, which has an additional column for storing the confidences for timely prefetching actions, i.e. the deltas. The top n confident timely deltas are selected as candidates for prefetch and the respective prefetches are issued, while also allocating entries in the TT to track these decisions that were made.

2.4 Detailed Overview

This section provides a detailed overview on the *Tracking Table* (TT) and the *Extended Pattern Table* (EPT) as they are the core components that drives R-SPP, which was briefly explained in section(2.3).

2.4.1 Tracking Table

Tracking Table (TT) is a fully-associative/set-associative [7] table, whose objective is to track the *most-recent* prefetching decisions made, so as to appropriately *reward* them in the future. The exact structure of the table is shown in fig(2.5). The table is referred to, on both the phases of R-SPP: *issuing rewards* and *issuing prefetches*, each of which are explained below.

Suppose a demand request for cache-line (or block) b_i in page P_i has arrived. This marks the beginning of *issuing-rewards* phase. As explained already in section(2.3), if any of the prefetch requests, that is currently tracked in the TT, was responsible for bringing (P_i, b_i), then the EPT is referred to with the signature of the corresponding TT's entry and the confidence of the delta (also obtained from the corresponding TT's entry) in the EPT is given a reward depending on the whether or not the prefetch was a timely prefetch, which is found out by the following check

$$t_{prefetch} + \theta_{late} \leq t_{curr} \leq t_{prefetch} + \theta_{early} \tag{2.5}$$

where $t_{prefetch}$ and t_{curr} are the CPU cycle numbers of when the prefetch was issued and current cycle number respectively. θ_{late} and θ_{early} are *dynamically* changing thresholds for late and early prefetches respectively. θ_{late} is basically an estimate on the number of CPU cycles it takes for a block to be brought into the cache and θ_{early} is an estimate on the lifetime of a block (how long it stays in the cache) in CPU cycles.

An example of how the TT operates in the first phase is shown in fig(2.5). So, this will encourage the same action (i.e. delta) to be picked/avoided in the future, when the same signature is encountered again.

Along with this, each entry is also checked if they have also issued prefetches inside the same page P_i. If yes, then they are checked if they could have also brought b_i in a timely

[7] Currently implemented as a fully-associative table

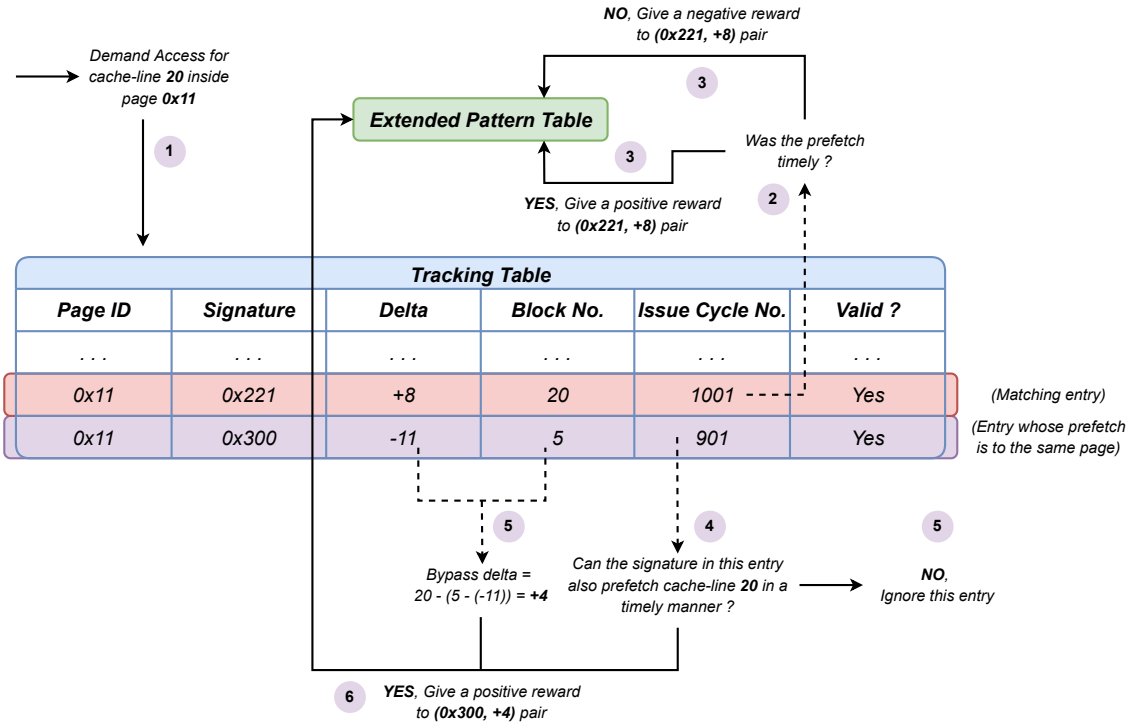


Figure 2.5: Detailed overview on how Tracking Table operates in the *issuing rewards* phase.

manner by comparing the time when their prefetches was sent with the current cycle number via eq(2.5). If no, then they are ignored. Else the delta, say d_{bypass}, that will point to b_i is calculated though the following equation

$$d_{bypass} = b_i - (b_{prefetch} - d_{prefetch}) \tag{2.6}$$

$b_{prefetch}$ is the block whose prefetch request was sent previously and $d_{prefetch}$ is the delta that was used to add to some *unknown* block to yield $b_{prefetch}$. Subtracting gives the (past) demand requested block number (now *known* to the prefetcher) and finally, subtracting it from b_i gives the delta that is required. The EPT is then referred to, using the corresponding entry's signature and confidence for the action d_{bypass} is updated, i.e. given a positive reward. The point is, whenever the signature, that was previously used to prefetch block $b_{prefetch}$ is encountered again in the future, it is also encouraged to prefetch b_i, provided that the confidence goes high enough. The operation for this *bypass* mechanism is also shown in fig(2.5). After every entry is checked similarly, it marks the end of *issuing rewards* phase.

2.4.2 Extended Pattern Table

Once *issuing rewards* phase ends, it marks the beginning of *issuing prefetches* phase. The signature of the corresponding page is extracted from the *Signature Table* of SPP [11], it is then used to query the EPT. The top n confident deltas, say $d^*_{1\cdots n}$ from the *timely* column are picked as the prefetch candidates and prefetches are sent to all the blocks $b^*_{1\cdots n}$ (that fall inside P_i) such that

$$b^*_j = b_i + d^*_j, \quad \forall j \in \{1 \cdots n\} \tag{2.7}$$

Also n entries in the TT are allocated to track these prefetching decisions, so that they can be appropriately rewarded in the future.

2.5 Evaluation

2.5.1 Simulation Infrastructure and Benchmarks

The evaluation of R-SPP was done on ChampSim, which is a trace-based simulator, that has been used as the go-to simulation tool for evaluating state-of-the-art prefetcher proposals in 2nd, 3rd data-prefetching championship (DPC-2 and DPC-3) and even the machine-learning based data-prefetching competition organized by ISCA 2021. Single-core simulations were performed with a warmup of 40M instructions and the results obtained were from the next 100M instructions. The performance of R-SPP is compared against a baseline (a system with no prefetching) and SPP.

The benchmarks used for evaluation currently, come from the widely used SPEC2017 benchmark suite. In particular, 45 memory-intensive benchmarks (for which last-level cache's MPKI ≥ 1) [14] from the suite were used for the evaluation, which are publicly made available for academic research, from the DPC-3's website.

2.5.2 Results

Fig(2.6) plots the normalized performance of both the prefetchers, when compared against a baseline with no prefetching and Table(2.1) summarizes the results. Fig(2.7) plots the accuracy of the corresponding prefetchers and table(2.2) summaizes the results. Fig(2.8) plots the amount of late prefetches issued by the corresponding prefetchers and table(2.3) summarizes the results.

Benchmark	**Prefetcher**		
	SPP (without look-ahead)	**SPP**	**R-SPP**
SPEC2017	30.541 %	45.673 %	47.056 %

Table 2.1: Performance improvement on SPEC2017 benchmarks

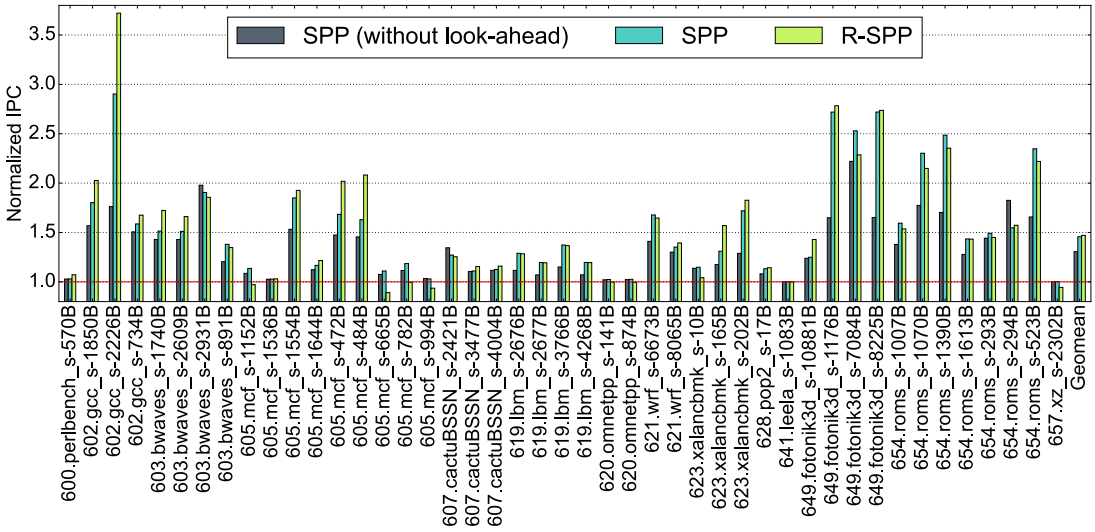


Figure 2.6: Normalized IPCs on SPEC2017 benchmarks, when compared against a baseline with no prefetching. On a geometric average, SPP [11] obtained an improvement of **30.541 %** (without look-ahead), **45.673 %** (with look-ahead) and R-SPP obtained an improvement of **47.056 %**.

Benchmark	**Prefetcher**		
	SPP (without look-ahead)	**SPP**	**R-SPP**
SPEC2017	89.610 %	90.462 %	47.743 %

Table 2.2: Prefetch Accuracy on SPEC2017 benchmarks

Benchmark	**Prefetcher**		
	SPP (without look-ahead)	**SPP**	**R-SPP**
SPEC2017	21.335 %	5.112 %	0.751 %

Table 2.3: Amount of late prefetches issued on SPEC2017 benchmarks

2.5.3 Prefetcher Configurations

For the sake of analyzing the best performance that can be obtained from both the prefetchers in their (almost) ideal conditions, storage overhead was not taken into consideration for these experiments, i.e. it is assumed that there is unlimited storage budget available for both the prefetchers, for the time being. Since R-SPP builds upon SPP [11], *Global History Register* module was turned off for SPP [11], as it was not implemented for R-SPP.

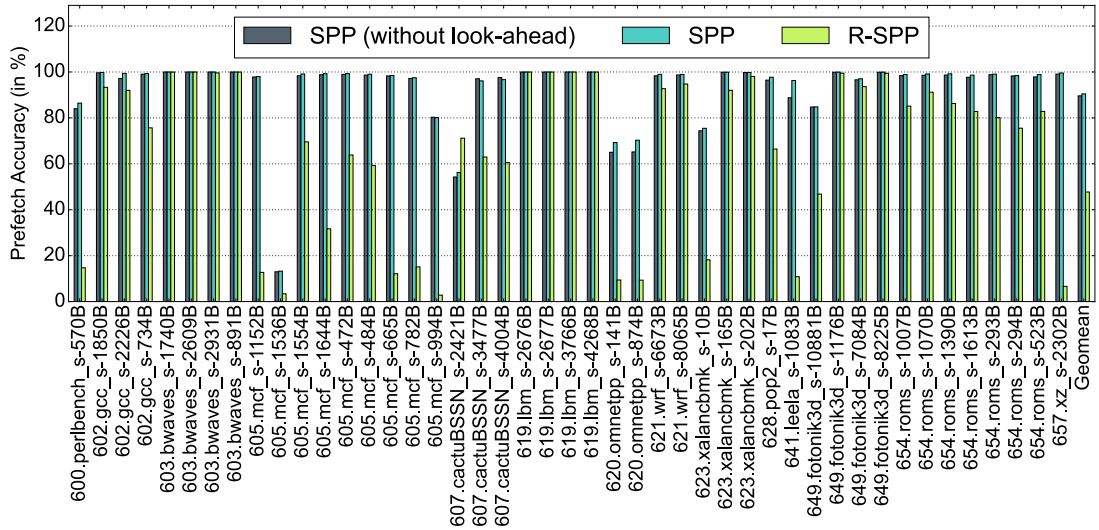


Figure 2.7: Prefetch accuracy (in %) on SPEC2017 benchmarks. On a geometric-average, SPP [11] obtained an accuracy of **89.610 %** (without look-ahead), **90.462 %** (with look-ahead) and R-SPP obtained **47.743 %**.

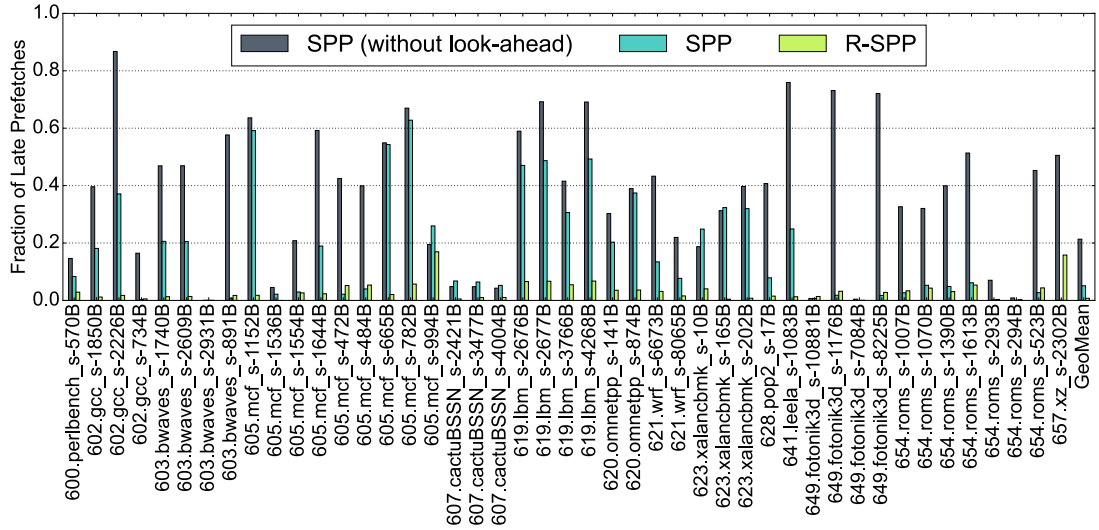


Figure 2.8: Amount of late prefetches issued by SPP [11] (with/without look-ahead) and R-SPP on SPEC2017 benchmarks. On a geometric average, SPP [11] issued **21.335 %** of late prefetches (without look-ahead), **5.112 %** and R-SPP issued only **0.751 %**.

Module	# of Sets	# of Ways
Signature Table	1	1024
Prefetch Filter	1024	1
Pattern Table	4096	127
Tracking Table	1	1024
Extended Pattern Table	4096	127

Table 2.4: Prefetcher configurations for both SPP [11] and R-SPP. First two modules are common for both the prefetchers. Pattern Table is exclusive to SPP [11] and the remaining two modules are exclusive to R-SPP

Chapter 3

Neural-Filter

3.1 Motivation

Although *Reinforced Signature-Path Prefetcher* (R-SPP) improved the performance (+0.948 % over SPP, on a geometric-average) and issued extremely less late-prefetches (0.751 %), as was described in section 2.5, it was found to be less *accurate* (47.743 %) when compared to SPP (90.462 %). Although late-prefetches impact performance, not *all* late-prefetches are actually harmful - as long as some fraction of the transfer latency is hidden, it is good enough. This can be observed from majority of the prefetchers ([2], [3], [11], [13], [14] etc.) that do not consider lateness of prefetches, but nonetheless improve the performance significantly. But *accuracy* of the prefetchers, which is defined as,

$$\text{Prefetch Accuracy} = \frac{\text{No. of Useful Prefetches}}{(\text{No. of Useful + No. of Useless Prefetches})} \tag{3.1}$$

play a significant role in impacting the performance, especially under *low* memory bandwidth because the miss latencies are relatively much higher and a (useless) prefetch evicting a potentially useful cache-line (that will be used nearby in the future) would negatively impact the performance much higher than it would have, under high memory bandwidth.

Fig(3.1) and fig(3.2) empirically shows the impact on performance caused due to accuracy of prefetchers under low memory bandwidth on SPEC2017 and GAP benchmarks. For the case of SPEC2017 benchmarks, when the DRAM bandwidth was 100 MT/s, although all the prefetchers degraded the performance (baseline was a system with *no* prefetcher), SPP [11] degraded the performance the least (by $\approx$ 4 %), followed by Bingo [2] (by $\approx$ 10 %) and IPCP [14] (by $\approx$ 20 %) - the order being inverse of the prefetch accuracy. For the case of GAP benchmarks, it is similar, but the performance impact is even more significant - SPP [11] degrading the performance by $\approx$ 6 %, Bingo [2] by $\approx$ 40 % and IPCP [14] by $\approx$ 50 %.

This motivated the the need for a framework that can also improve the accuracy of the prefetchers alongside timeliness, which can be augmented with *any* prefetcher and hence, the goal shifted from improving the performance by minimizing the late-prefecthes towards

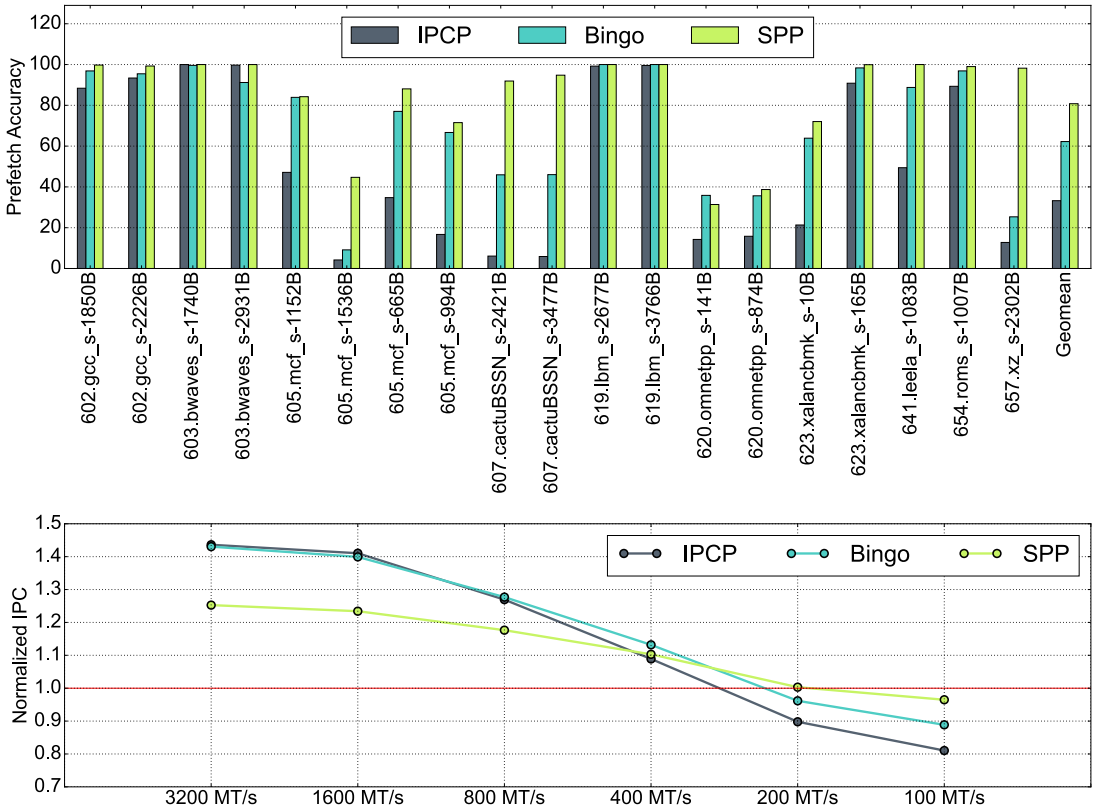


Figure 3.1: On a geometric average, the accuracy was **33.242 %** on IPCP [14], **62.232 %** on Bingo [2] and **80.792 %** on SPP [11] prefetcher on the SPEC2017 benchmarks used above. The figure below the accuracy plot describes the change in performance relative to a baseline-system with *no* prefetcher, under varying DRAM bandwidth.

minimizing the negative impact caused by the prefetchers due to their bad prefetching decisions by filtering out such requests.

3.2 Model Architecture

Fig(3.3) describes the high-level architecture of Neural-Filter (NeuFi) model. First, a *dense* representation of the current system-state is extracted from the current memory-access behavior, then along with the cache-line to prefetch as well the cache-level (till which it will be prefetched) and the current page access-vector (described in section 3.3.2) is fed to the prediction networks, which predicts if the the prefetched cache-line will be timely and useful. Following subsections describe each component of NeuFi in detail.

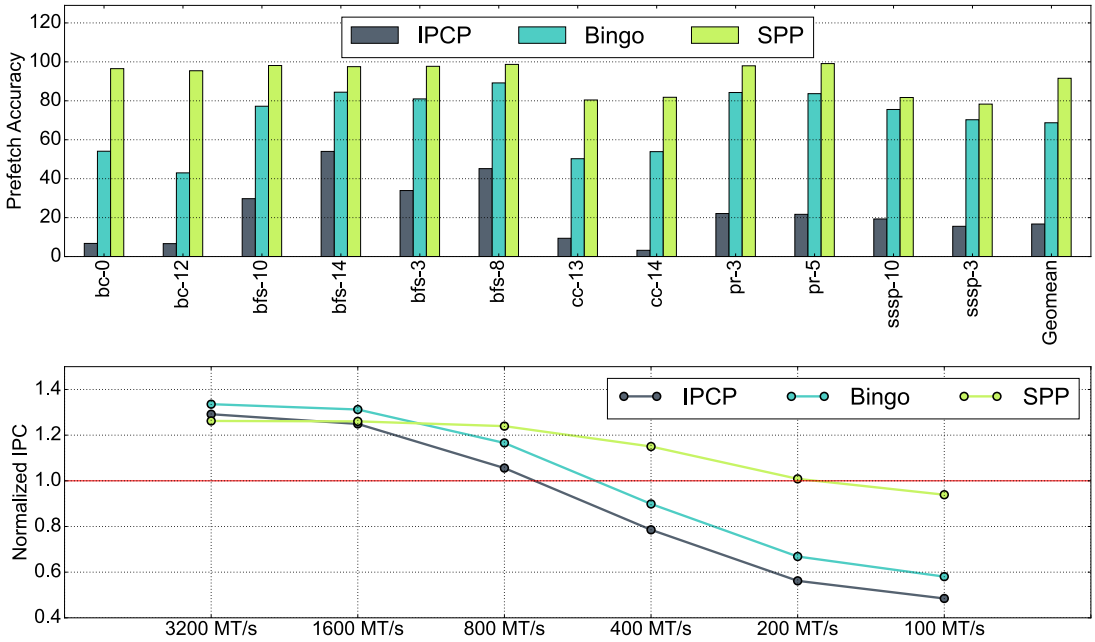


Figure 3.2: On a geometric average, the accuracy was **16.721 %** on IPCP [14], **68.716 %** on Bingo [2] and **91.564 %** on SPP [11] prefetcher on the GAP benchmarks used above. The figure below the accuracy plot describes the change in performance relative to a baseline-system with *no* prefetcher, under varying DRAM bandwidth.

3.2.1 Byte-Distributed Embedding with Attention

Consider a 128-bit binary vector V that is used to represent the deltas seen within a page P or observed by an instruction-pointer (IP) I. For example, if deltas $-1, +1$ and $+2$ were seen within P or observed by I, then the corresponding bits of those deltas within V would be set to 1 and everything else would be 0s, i.e.

$$V = [0, 0, \cdots, 0, 1, 0, 1, 1, 0, \cdots, 0]^T$$

Because an *embedding* is nothing but a lookup-table, a naive-implementation would require 2^{128} entries to account for all possibilities of vector V, which is computationally infeasible. One way to mitigate this is to keep entries for only those vectors V' such that V' was observed frequently within the dataset and group the remaining ones as a single entity V'' (unknown) - a technique frequently applied in Natural-Language-Processing (NLP) tasks. This is however, not applicable in the context of the above mentioned page-delta or IP-delta vectors V because of different behaviors introduced by different application workloads (per-page and/or per-IP) and depending on the cache-level, requests getting filtered out by higher-levels of the cache, leading to more "variety" of the delta vectors V. Moreover, there

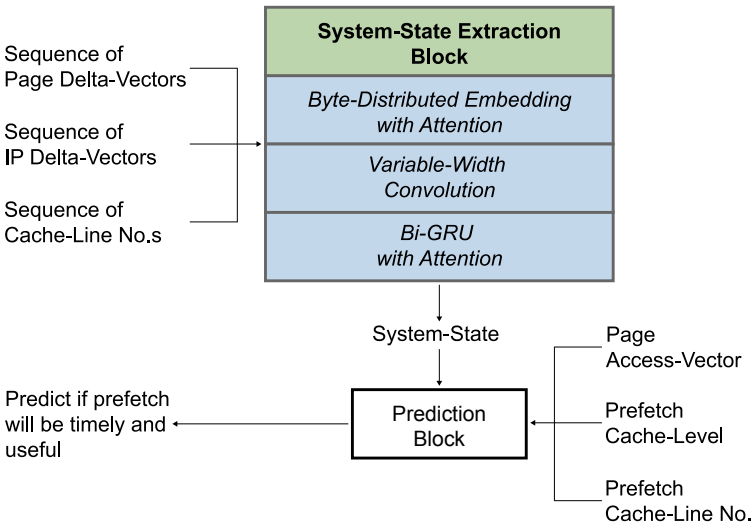


Figure 3.3: High-level architecture of Neural-Filter (NeuFi)

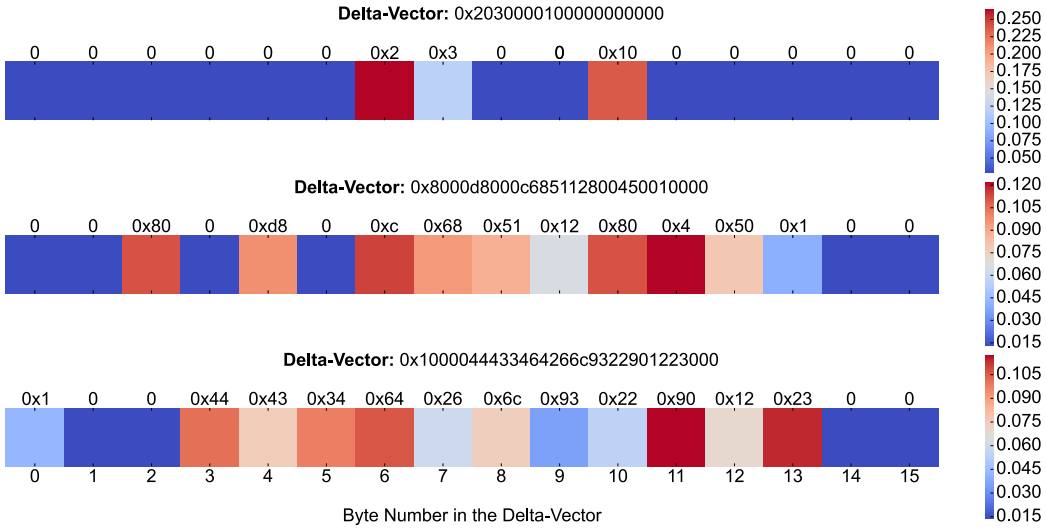


Figure 3.4: Attention weights being given to the bytes of randomly sampled page delta-vectors of `605.mcf_s-665B`

is the issue of observing delta-vectors during test-time that might not have been observed during model training.

To mitigate the above mentioned issues, this work proposes a technique called *Byte-Distributed*

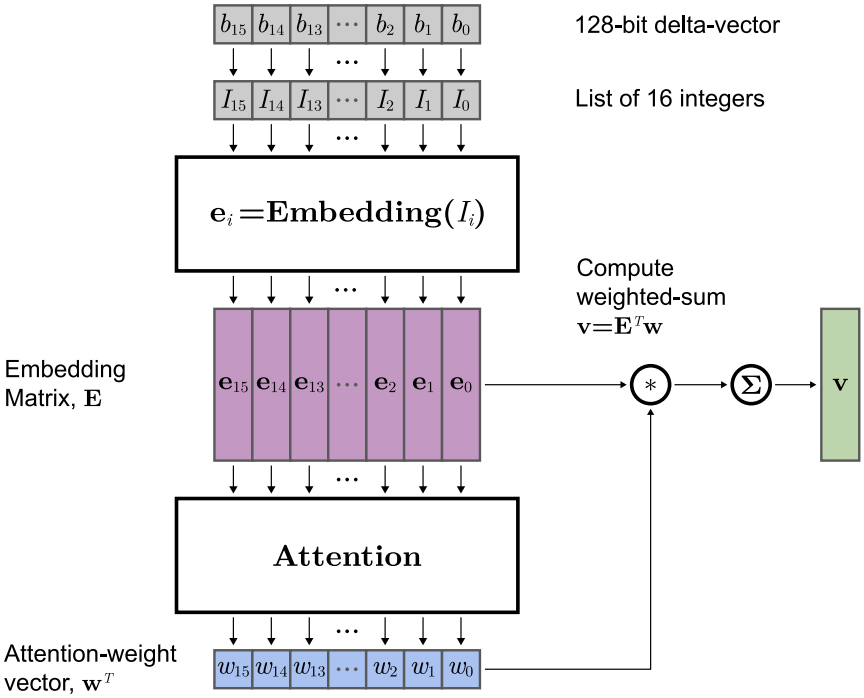


Figure 3.5: Working of BDEA

Embedding with Attention (BDEA), which is similiar to [12]. However unlike [12], this works with delta-vectors, which is a more general representation since bytes in an address do not say anything about *similarity*, especially when those addresses were not seen in the training set and that it uses *Attention* [1] to compute the final embedding vector. The working of BDEA [1] is described in detail in 1 and fig(3.5) provides a high-level overview.

The main reason for using Attention [1] is to ensure that only those bytes that are rich with deltas, i.e. majority of the bits set to 1, contribute to the final embedding vector - which is a weighted-sum of the individual byte-embeddings. Intuitively it would mean that bytes that are rich with deltas would be given relatively higher weights than the bytes that are not and the least weight being given to the bytes that are just zeros, which should make sense since a byte being zero indicates that none of the corresponding deltas within that byte were seen and hence should not be considered for the most part.

To prove that this is indeed happening practically, fig(3.4) shows the attention-weights being given to the bytes of few delta-vectors (picked at random) from a SPEC2017 benchmark - `605.mcf_s-665B`. Since an embedding of some element k produces a dense continuous vector

[1] It is to be noted that *positional-encoding* (introduced in [6]) has not been used to encode respective byte-positions into the delta-vector that is split into bytes, as described in 1. Whether or not it will improve performance, is a work left for future.

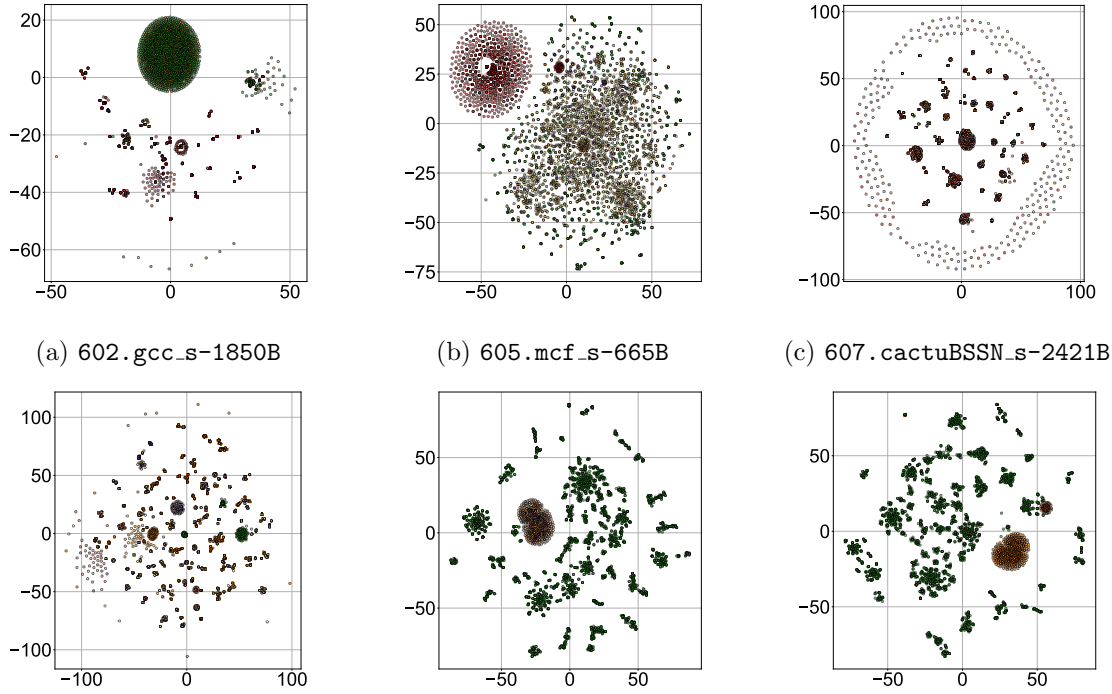
Algorithm 1: Byte-Distributed Embedding with Attention (BDEA)

Function BDEA (V, E, A);
Input : 128-bit Delta-Vector D,
Embedding-Network E,
Attention-Network A
Output: Embedding of D, say V
$i \leftarrow 0$;
$V_E \leftarrow 0$;
$E \leftarrow \phi$;
$S \leftarrow \phi$;
repeat
    // Extract the 8 components of binary-vector D
    // Then convert that binary-vector into an integer
    $d_i \leftarrow D[i : i + 8]$;
    $x_i \leftarrow$ binary_to_integer(d_i);
    // Get the embedding vector for x_i and corresponding attention-score
    $e_i \leftarrow E(x_i)$;
    $s_i \leftarrow A(e_i)$;
    Enqueue e_i to E ;
    Enqueue s_i to S ;
    // Move to next 8 components
    $i \leftarrow i + 8$;
until $i \geq |V|$;
// Compute attention-weights $\in [0, 1]$ and finally, the weighted-sum
$W \leftarrow Softmax(S)$;
$V \leftarrow W^T E$;
return V;

such that the embedding of elements that are similar to k are *closer* to k in the embedding's vector-space, there is a need to check whether the final embedding of the delta-vectors indeed have this property. To verify this, N page delta-vectors were sampled at random from the datasets of corresponding SPEC2017 and GAP benchmarks and their embeddings were plotted (after projecting onto 2D space using T-SNE), the results of which are show in fig(3.6).

Consider the five highlighted clusters from the embedding plot of `607.cactuBSSN_s-2421B` from fig(3.6c) as shown in fig(3.7). Since the page delta-vectors are bit-vectors, to check for similarity, *hamming-distance* measure is used between the vectors in each of the clusters and the results of the analysis are shown in table-3.1.

The median hamming-distance increases as the distance between two clusters increases and it is the least when compared against the delta-vectors of the same cluster - empirically



(a) `602.gcc_s-1850B` (b) `605.mcf_s-665B` (c) `607.cactuBSSN_s-2421B`

(d) `641.leela_s-1083B` (e) `pr-5` (f) `sssp-10`

Figure 3.6: Visualization of embeddings of N (= 5000) page delta-vectors (sampled at random) colored according to instruction-pointers (IPs), from some SPEC2017 and GAP benchmarks, using T-SNE

Cluster ID	C_1	C_2	C_3	C_4	C_5
C_1	**0**	9	1	13	19
C_2	-	**0**	8	4	10
C_3	-	-	**0**	12	18
C_4	-	-	-	**0**	7
C_5	-	-	-	-	**5**

Table 3.1: Median hamming-distance between the vectors of corresponding clusters

proving the similarity property and the working of BDEA. Although the distance between clusters C_1 and C_3 appears to be higher than the clusters C_1 and C_2, they must have been closer in some other dimensions but due to dimensionality reduction, it got lost - which probably explains why the median hamming-distance between the vectors of C_1 and C_3 is lower than the vectors of C_1 and C_2.

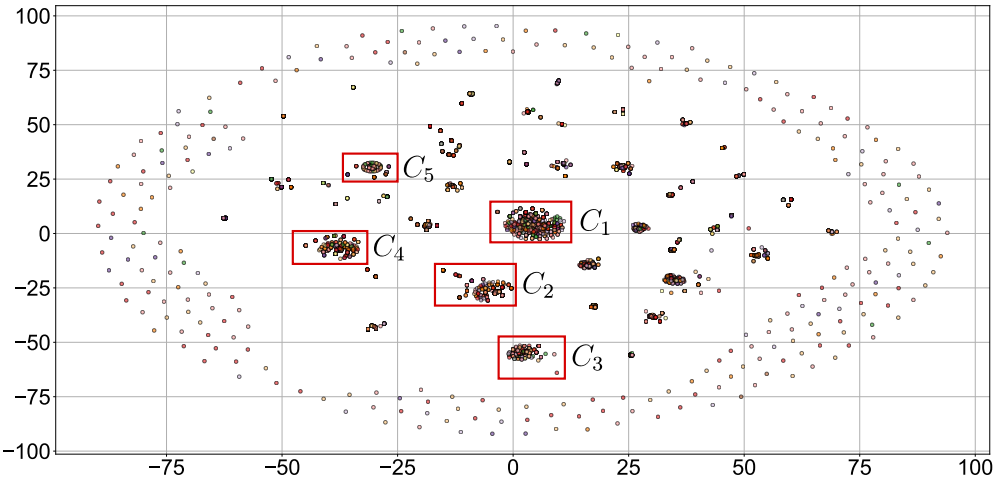


Figure 3.7: Clusters of interest for the analysis of delta-vectors in `607.cactuBSSN_s-2421B`

3.2.2 Variable-Width Convolutions

Access patterns repeat temporally - the amount being dependent on an application or how often a specific region of an application is executed. For example, an application that frequently accesses the elements of an array sequentially would have an repetitive access pattern whenever the array access is initiated. Consider the `602.gcc_s-2226B` SPEC2017 benchmark which sees deltas of +1 and -1 (in terms of cache-lines) most of the time, as shown in fig(3.8) - which is a good indicator for the presence of patterns that are repetitive temporally. To verify if this is actually the case, a temporal window of 32 sequential accesses T was picked from the benchmark's dataset and temporal windows T' ($|T'| = |T|$) that were *similar* to T were searched throughout the dataset - four of which are visualized in fig(3.9), according to their page access-vectors. A page access-vector p is a 64-bit [2] binary vector such that $p_i = 1$ if i'th cache-line within the page was accessed. Since it is a well known fact that any convolutional *kernel* K (in the context of images) which learnt to detect feature f can detect f irrespective of wherever it occurs spatially within an image, similar line of reasoning can be applied to 1D temporal sequences to detect features that repeat temporally - which is what this work utilizes. To be more specific, it uses *Variable-Width Convolution* bank (VWC), similar to the convolution bank used in [17], which is basically using n 1D convolution kernels [3] K_i where,

$$WindowSize(K_i) = i, \quad \text{such that } 1 \leq i \leq n \tag{3.2}$$

[2] ChampSim simulator uses 4KB pages and 64 byte cache-lines, which means there are 4096/64 = 64 cache-lines per-page.

[3] Not 2D because we are not dealing with images but 1D temporal sequences

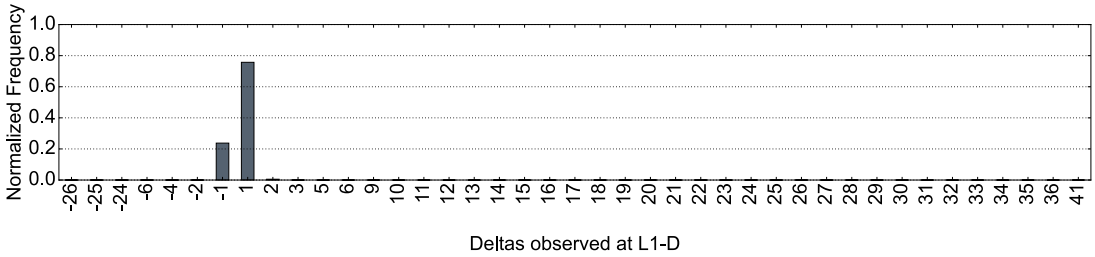


Figure 3.8: Normalized frequency of deltas observed at L1-D on `602.gcc_s-2226B`

The working of VWC layer is described in fig(3.10). At a high-level, it takes a temporal sequence of T *features* F_{ip} [4] and runs each 1D kernel K_i on F_{ip} individually to produce outputs o_i , before concatenating the each o_i together and finally running them through a *channel-compression* 1D kernel (of window size 1) to produce the final output sequence F_{op}.

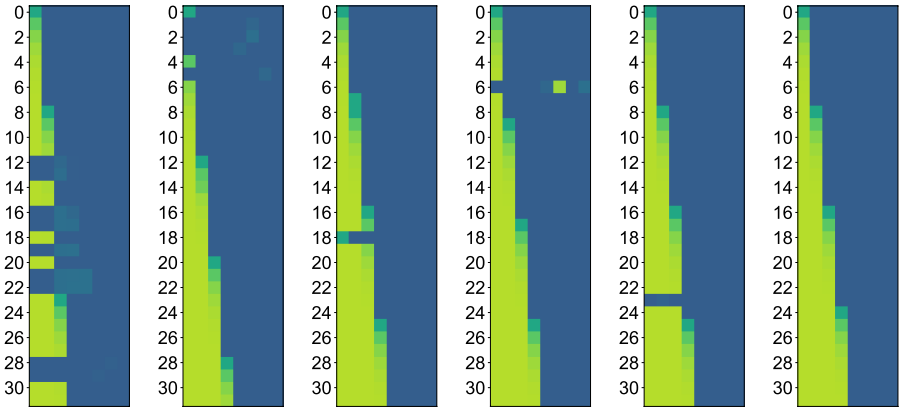


Figure 3.9: Visualization of similar access patterns of `602.gcc_s-2226B` observed across different parts of the benchmark's dataset

3.2.3 Bi-GRU with Attention

Long Short-Term Memory networks (LSTMs) [10] have been empirically proven to model sequences of arbitrary lengths and have been used in works similar to this work, for example, [16]. Although in theory they can model sequences of any length, it starts to lose

[4] To be more specific, the input to a VWC layer is a temporal sequence of embeddings

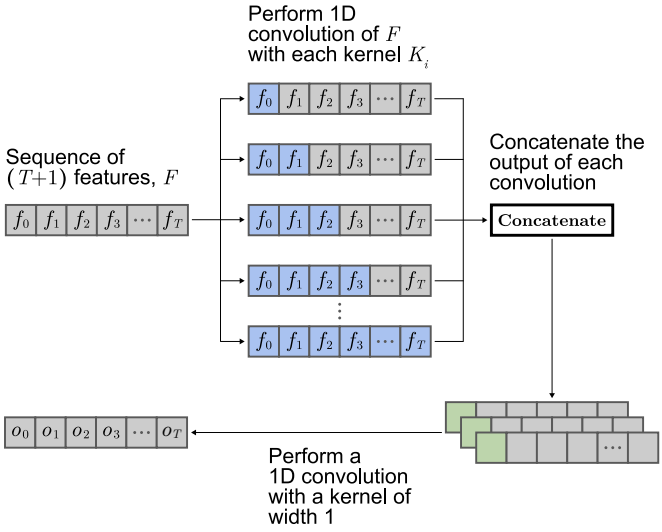


Figure 3.10: Working of VWC layer

effectiveness beyond a specifc sequence length, as was also mentioned in [10]. One way to mitigate this issue is usage of *bidirectional* LSTMs (Bi-LSTMs) which utilizes two LSTMs - one operating normally and the other operating on reversed version of the input sequence before concatenating the outputs of both LSTMs together as the final output. Although this mitigated the issue by some amount and allowed for much longer sequences, it still was limited by the sequence length. *Attention* mechanism [1] removed this restriction by storing intermediate state outputs per time-step and later on, taking a weighted sum of those intermediate outputs - giving more weights to those output states that are more relevant. Gated Recurrent Units (GRUs) [7] are similiar to LSTMs, but with fewer parameters and perform similar or better than LSTMs in certain tasks as shown in [8]. This work utilizes Bi-GRU with Attention mechanism to generate the final context vector. At a high-level, it takes the sequence of output features produced by the *Variable-Width Convolution* (VWC) layer (described previously in section 3.2.2) and produces an intermediate sequence of output states before taking a weighted sum via the Attention mechanism, to produce the final context vector $v_{context}$.

3.2.4 Prediction Feed-Forward Networks

Once the context vector $v_{context}$ is obtained from the Bi-GRU (as described in section 3.2.3), it is concatenated with an embedding of a block b (say b_{embed}) to which a prefetch request is issued by the *black-box* prefetcher. Along with this, the current page-access vector p (described previously in section 3.2.2) is also concatenated to produce the input I_{ffn}.

$$I_{ffn} = [v_{context};\ p;\ b_{embed}]^T$$

I_{ffn} is fed to two *separate* feed-forward networks for predicting whether the prefetch would be *timely* [5] and *useful*. Each prediction network has one unit in the output layer, which predicts the probability y_{prob} of being Yes ($y_{prob} = 1$) or No ($y_{prob} = 0$). For all the experiments in this work, thresholds for both the networks were set to 0.8, beyond which the prediction would map to Yes, i.e. if $y_{pred} \geq 0.8$, the network would predict Yes.

3.3 Model Configuration and Dataset

3.3.1 Model Configuration

Table(3.2) describes the configuration for embedding part of the model including *Byte-Distributed Embedding with Attention*(BDEA), table(3.3) describes the configuration of *Variable-Width Convolution* (VWC) layer, table(3.4) describes the configuration of Bi-GRU with Attention and finally, table(3.5) describes the configuration of the two prediction feed-forward networks.
Each model was trained for 100 epochs using *Adam* optimizer with a learning-rate of 10^{-3} and *binary cross-entropy* loss function. The sequence-length was set to 9, i.e. apart from current access to the cache, 8 prior accesses were used as the context for current one.

Input	**Embeddding Type**	**Total Embeddings**	**Embeddding Dim.**
Page Delta-Vector	BDEA	256	64
IP Delta-Vector	BDEA	256	64
Cache-Line #	Naive	64	64

(a) Embedding configuration of the inputs used to extract the system-state

Input Dim.	**Hidden Units**	**Output Dim.**
64	[32]	1

(b) Attention network configuration of BDEA. Each hidden-layer is followed by a ReLU activation.

Table 3.2: Embedding layer configuration

[5]The data extraction API, described in section 3.3.2 has a parameter which controls the criterion for classifying the prefetches as timely or not. For all the experiments in this work, any prefetch that hid atleast 40% of the latency were classified to be timely and the models were trained using this.

Input	Input Channels	Output Channels	Output Channels (after reduction)	Range of Kernel Widths
Page Delta-Vector Embedding	64	64*8	64	[1, 8]
IP Delta-Vector Embedding	64	64*8	64	[1, 8]
Cache Line # Embedding	64	64*8	64	[1, 8]
Each of (above 3) Outputs from VWC	192	192*8	192	[1, 8]

Table 3.3: Configuration of VWC layers of each input. Each convolution is followed by a max-pool layer of width 3 and stride of 1. The channel-reduction layer uses kernels of width 1 to reduce the output size.

Input Dim.	# of Layers	Output Dim.
192	2	64

(a) Bi-GRU configuration

Input Dim.	Hidden Units	Output Dim.
64	[32]	1

(b) Attention network configuration of Bi-GRU. Each hidden-layer is followed by a ReLU activation.

Table 3.4: Bi-GRU with Attention configuration

3.3.2 Dataset for Training, Validation and Testing

The datasets used for training, validation and testing were extracted from SPEC2017 and GAP benchmarks *per-prefetcher*, using an API that was written for ChampSim, made open-source [6]. The configuration of the simulation systems were set to *default* ChampSim configurations, which included DRAM bandwidth being 3200 MT/s. Table 3.6 describes each line in the dataset.

[6]The data extraction API can be found in this GitHub repository

Input Dim.	Hidden Units	Output Dim.
(2*64) + 64 + 64 + 1	[32, 64]	1

(a) Feed-forward network configuration for predicting *timeliness*. Each hidden-layer is followed by a ReLU activation. The input layer consists of output of Bi-GRU, cache-line embedding of the line to prefetch, page access-vector and the cache-level till which it will be prefetched.

Input Dim.	Hidden Units	Output Dim.
(2*64) + 64 + 64 + 1	[32, 64]	1

(b) Feed-forward network configuration for predicting *usefulness*. Each hidden-layer is followed by a ReLU activation. The input layer consists of output of Bi-GRU, cache-line embedding of the line to prefetch, page access-vector and the cache-level till which it will be prefetched.

Table 3.5: Prediction Networks (for timeliness / usefulness) configuration

Index	Item	Description
1	**Access-ID**	The access number of the cache, incremented at L1-D
2	**IP**	The instruction-pointer (IP) responsible for initiating the access
3	**Demand Address**	The demand-address (cache-aligned) with which the cache was queried
4	**Page Delta-Vector**	A 128-bit binary vector, where 1 indicates that the corresponding delta was seen in the page
5	**Page Access-Vector**	A 64-bit binary vector, where 1 indicates that the corresponding block was accessed in the page
6	**IP Delta-Vector**	A 128-bit binary vector, where 1 indicates that the corresponding delta was seen by an IP, within a page
7	**Prefetch-Vector**	A 64-bit binary vector, where 1 indicates prefetch to corresponding cache-line was sent
8	**Fill-Vector**	A 64-bit vector where 1 indicates fill till L1-D, 2 indicates fill till L2C and 4 indicates fill till LLC
9	**Timely-Vector**	A 64-bit binary vector, where 1 indicates that the corresponding prefetchd cache-line was timely
10	**Useful-Vector**	A 64-bit binary vector, where 1 indicates that the corresponding prefetchd cache-line was useful
11	**Delay-Vector**	A 64-bit vector which describes after how many accesses did demand-request arrive (negative for late prefetches, 0 also when demand-request never arrives)
12	**Candidate-Vector**	A 64-bit binary vector, where 1 indicates a future timely prefetch
13	**Median Miss-Latency**	An integer value indicating the median miss-latency (time required to bring a prefetch block, in no. of L1-D accesses)
14	**Median Eviction-Latency**	An integer value indicating the median eviction latency (in no. of L1-D accesses)

Table 3.6: Description of each line in the extracted prefetcher dataset

Chapter 4

Results

4.1 Misclassification Rate on Test Data

Fig(4.1) and fig(4.2) plots the misclassification rate obtained by running each trained model on corresponding benchmark's unseen test data and prefetchers. Table 4.1 summarizes the results obtained.

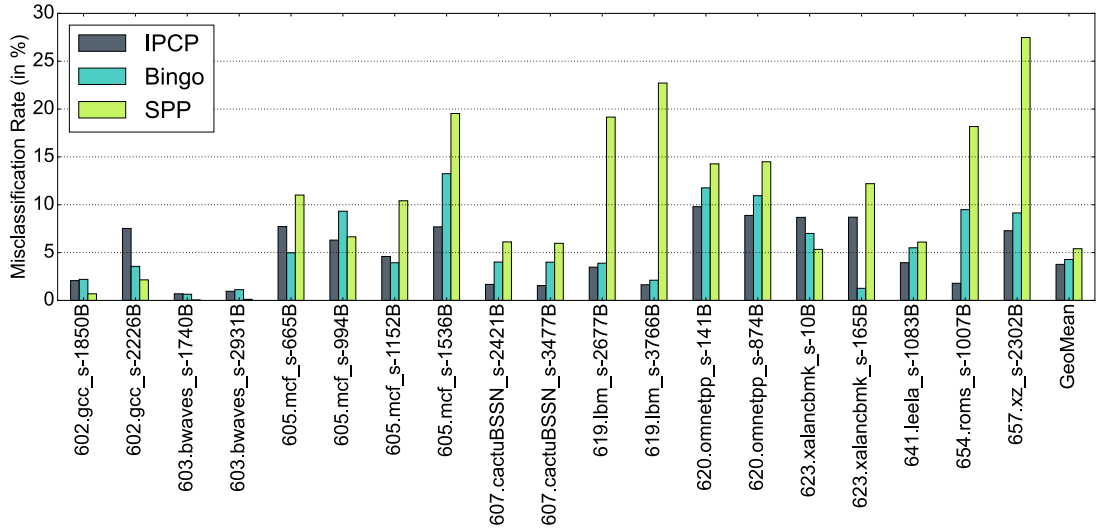


Figure 4.1: Misclassification rate by the models on various SPEC2017 benchmarks. On a geometric average, the misclassification rate was **3.767%** on IPCP [14], **4.277%** on Bingo [2] and **5.401%** on SPP [11] prefetcher

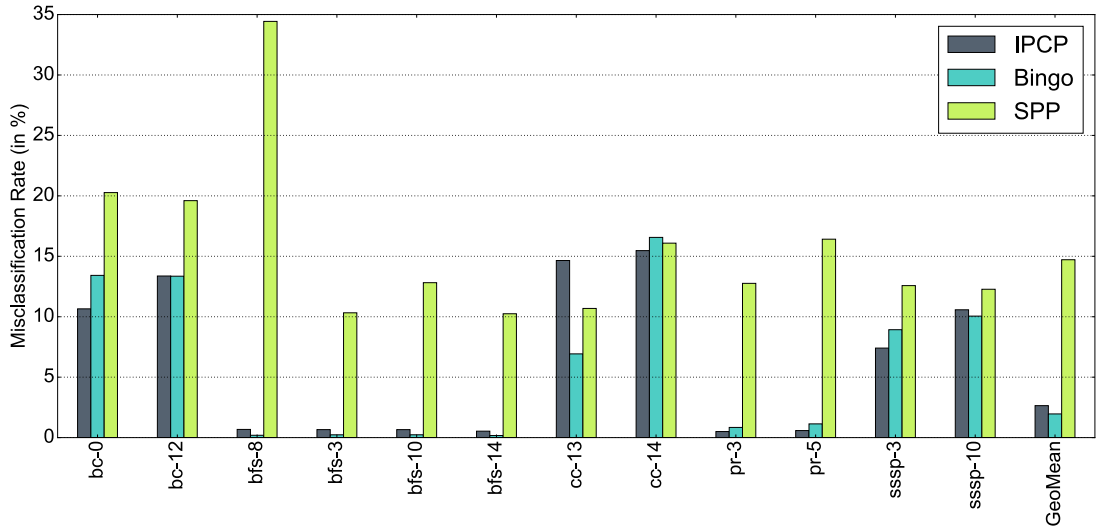


Figure 4.2: Misclassification rate by the models on various GAP benchmarks. On a geometric average, the misclassification rate was **2.645%** on IPCP [14], **1.959%** on Bingo [2] and **14.716%** on SPP [11] prefetcher

Benchmark	**Prefetcher**		
	IPCP	**Bingo**	**SPP**
SPEC2017	3.767 %	4.277 %	5.401 %
GAP	2.645 %	1.959 %	14.716 %

Table 4.1: Geometric-mean of the misclassification rates on unseen test data of SPEC2017 and GAP benchmarks

4.2 Late Prefetches Issued

Fig(4.1) and fig(4.2) plots the amount of late prefetches that were allowed to pass by NeuFi on corresponding benchmark's unseen test data. Table 4.2 summarizes the results obtained.

Benchmark	**Prefetcher**		
	IPCP	**Bingo**	**SPP**
SPEC2017	28.899 %	45.634 %	15.516 %
GAP	11.153 %	49.782 %	19.830 %

Table 4.2: Geometric-mean of the late prefetches allowed to pass by NeuFi on SPEC2017 and GAP benchmarks

Because some of the results in fig(4.3) and fig(4.4), for example `619.lbm_s-2677` (Bingo),

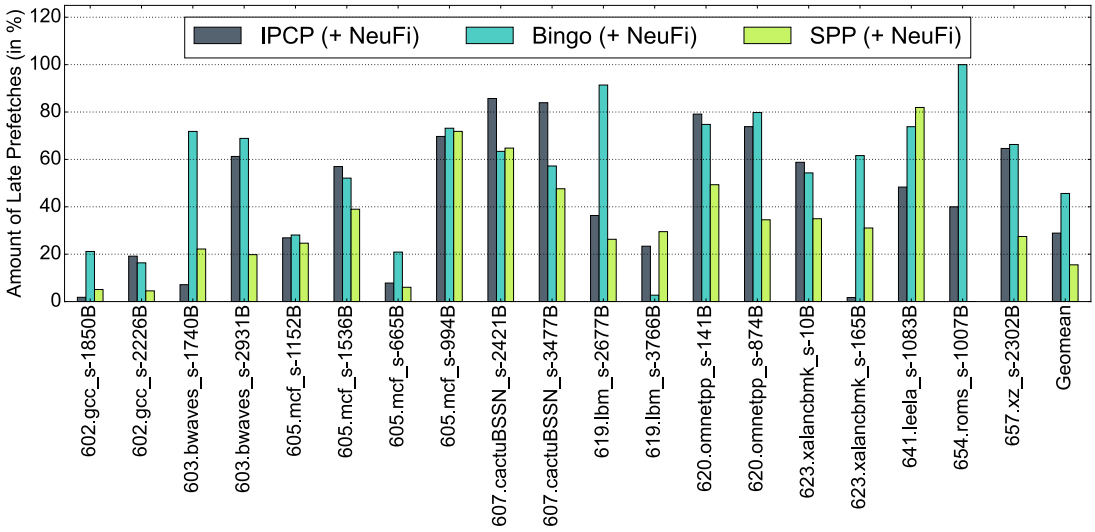


Figure 4.3: Amount of late prefetches allowed to pass by NeuFi on various SPEC2017 benchmarks. On a geometric average, the amount of late prefetches issued was **28.899%** with IPCP [14], **45.634%** on Bingo [2] and **15.516%** on SPP [11] prefetcher

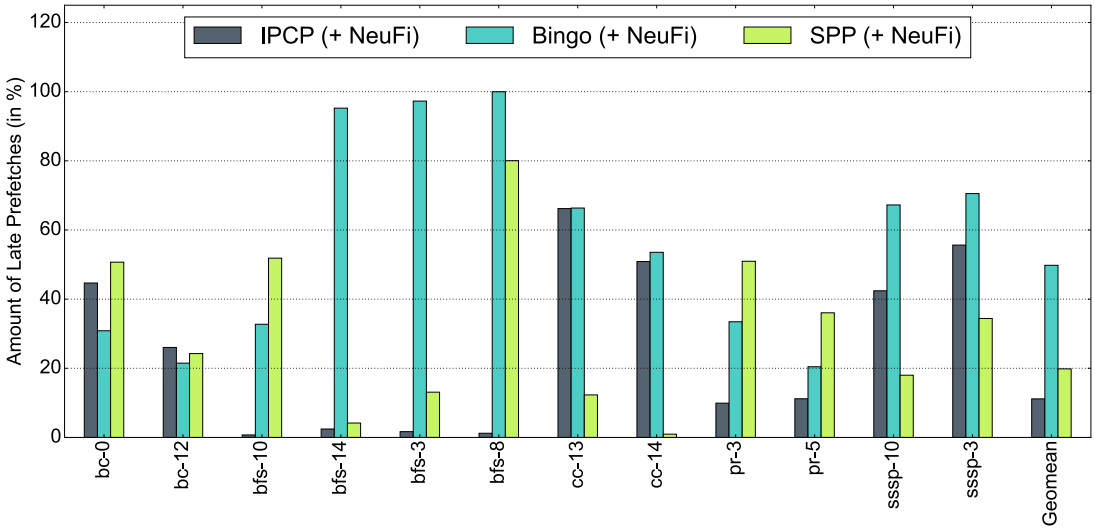


Figure 4.4: Amount of late prefetches allowed to pass by NeuFi on various GAP benchmarks. On a geometric average, the misclassification rate was **11.153%** on IPCP [14], **49.782%** on Bingo [2] and **19.830%** on SPP [11] prefetcher

`bfs-3` (Bingo) etc. show significant amount of late prefetches ($\geq 90\%$), some analysis is required on whether these happened because of model's inability to learn or because of some other cause(s) which was (were) not learnt by the model. Fig(4.5) and fig(4.7) plots the distance in no. of L1-D accesses (in $\log_2$ scale) after which the demand-access for those late prefetches arrived, whereas fig(4.6) and fig(4.8) plots a similar distribution, but for distances after which a prefetched cache-line is brought to the caches.
For the case of `619.lbm_s-2677` (Bingo), the plot in fig(4.6) shows that the median prefetch latency (in no. of L1-D accesses) is around $2^4 = 16$ L1-D accesses but the plot in fig(4.4) shows that the prefetches were late even though their demand-accesses arrived after around $2^6 = 64$ accesses. Similarly, for `bfs-3` (Bingo), the plot in fig(4.6) shows that the median prefetch latency is around $2^3 = 8$ L1-D accesses but the plot in fig(4.4) shows that the prefetches were late even though their demand-accesses arrived after around $2^6 = 64$ accesses. Similar reasoning can be applied to other such cases.
This can mean one of three things - The fact that the demand-accesses arrived after 64 accesses (median, in the above two cases) shows that although the prefetches were late, a significant amount of latency was hid or inability of the model to accurately learn the timeliness (bias in the corresponding datasets) or majority of the outliers are present the testing set.

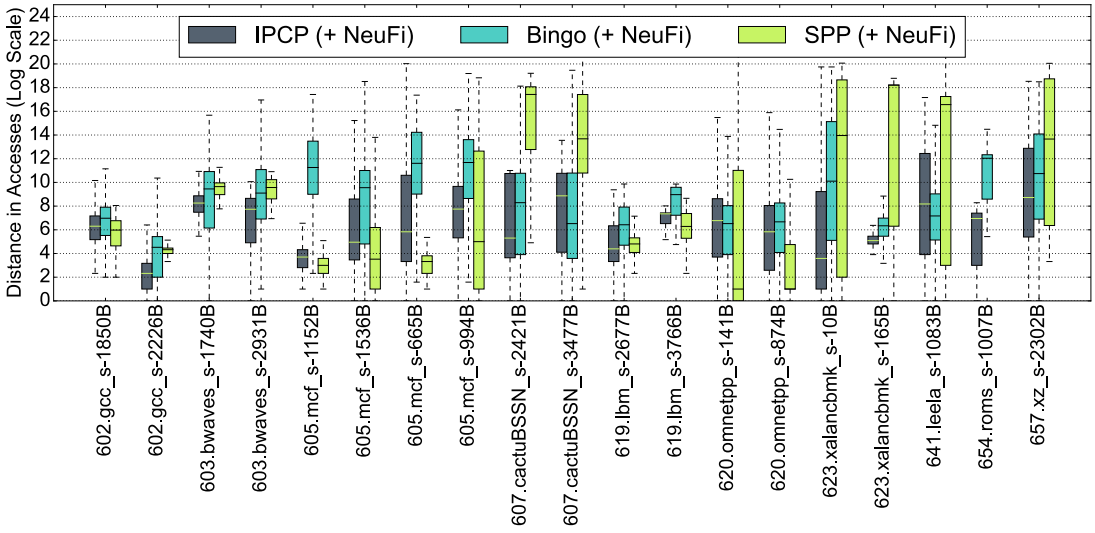


Figure 4.5: Box-plot denoting the distribution of distance (in $\log_2$ scale) in no. of L1-D accesses after which the demand-accesses for the late prefetches arrived, for SPEC2017 benchmarks

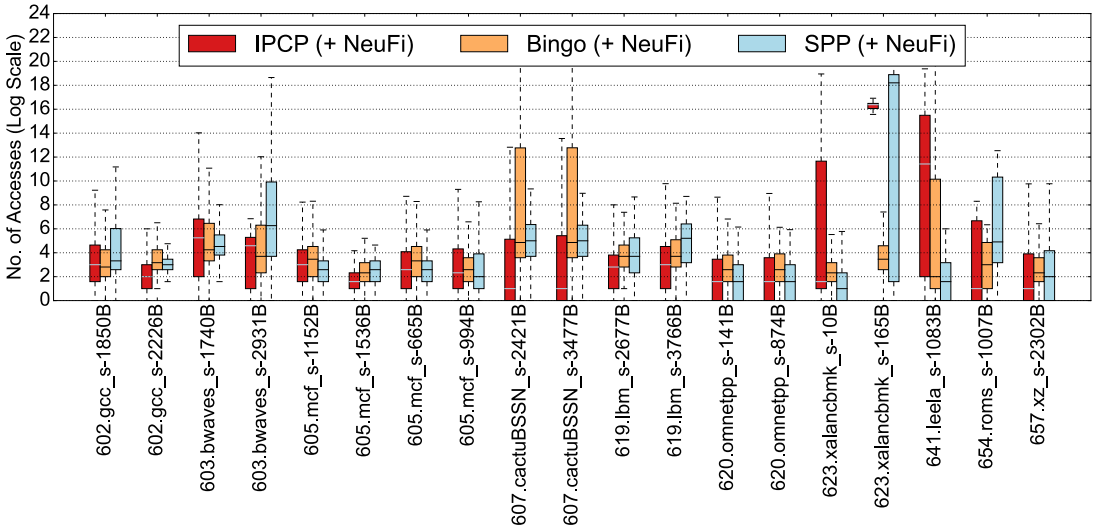


Figure 4.6: Box-plot denoting the distribution of distance (in $\log_2$ scale) in no. of L1-D accesses after which a prefetch is brought into the cache, for SPEC2017 benchmarks

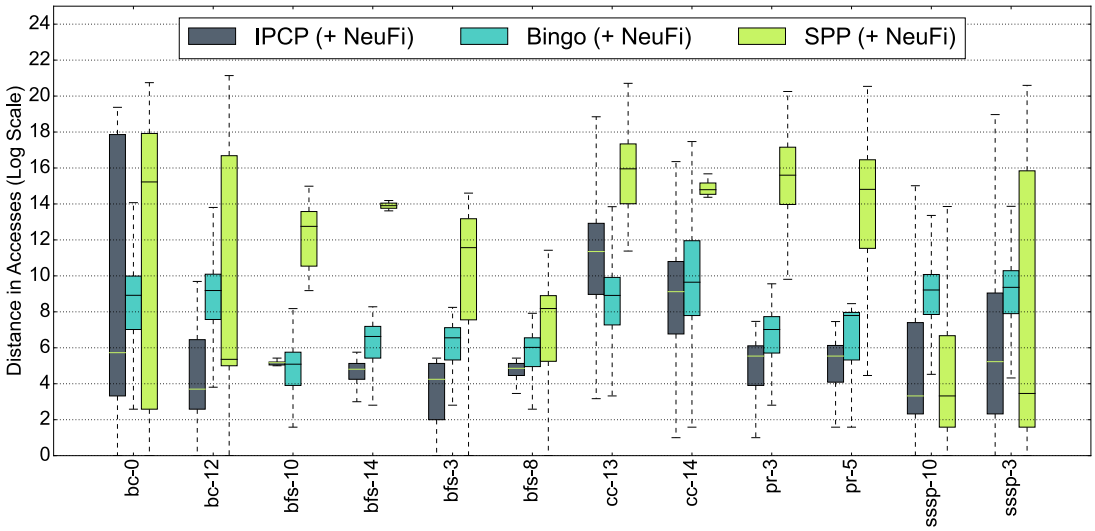


Figure 4.7: Box-plot denoting the distribution of distance (in $\log_2$ scale) in no. of L1-D accesses after which the demand-accesses for the late prefetches arrived, for GAP benchmarks

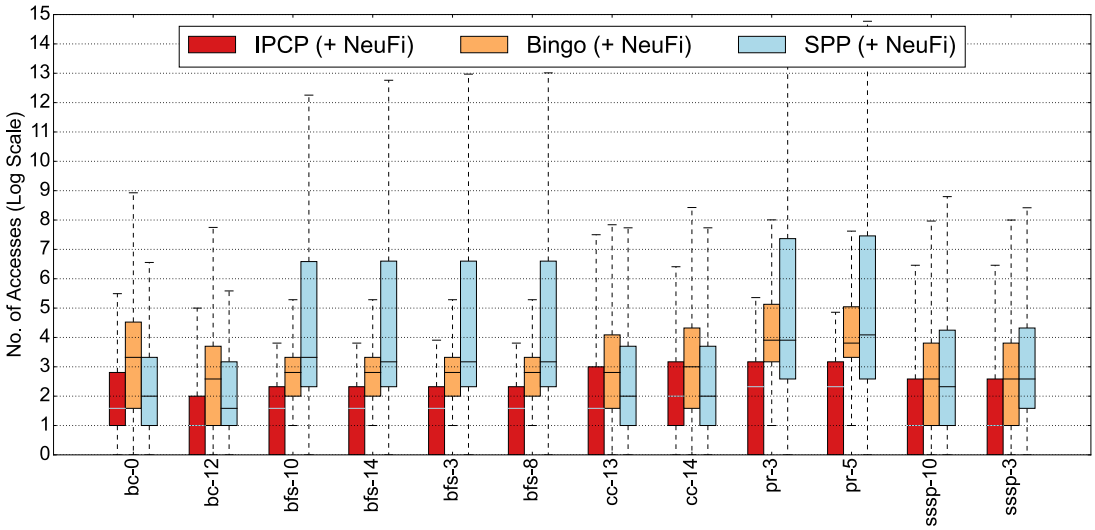


Figure 4.8: Box-plot denoting the distribution of distance (in $\log_2$ scale) in no. of L1-D accesses after which a prefetch is brought into the cache, for GAP benchmarks

4.3 IPC Improvement

Fig(4.1) and fig(4.2) plots the normlalized IPC (Instructions Per Cycle) of NeuFi augmented with the corresponding prefetchers. The baseline for each is the corresponding prefetcher without NeuFi. Table 4.2 summarizes the results obtained. Normalized IPC (in this context) is defined as

$$\text{IPC}_{\text{normalized}} = \frac{\text{IPC with NeuFi}}{\text{IPC without NeuFi}} \tag{4.1}$$

4.4 Reduction in Accesses to Off-Chip Memory

Fig(4.11a) and fig(4.11b) plots the normalized accesses to off-chip memory of NeuFi augmented with the corresponding prefetchers. The baseline for each is the corresponding prefetcher without NeuFi. Table 4.2 summarizes the results obtained. Normalized accesses is defined (similiar to eq(4.1)) is defined as

$$\text{Accesses}_{\text{normalized}} = \frac{\text{Accesses with NeuFi}}{\text{Accesses without NeuFi}} \tag{4.2}$$

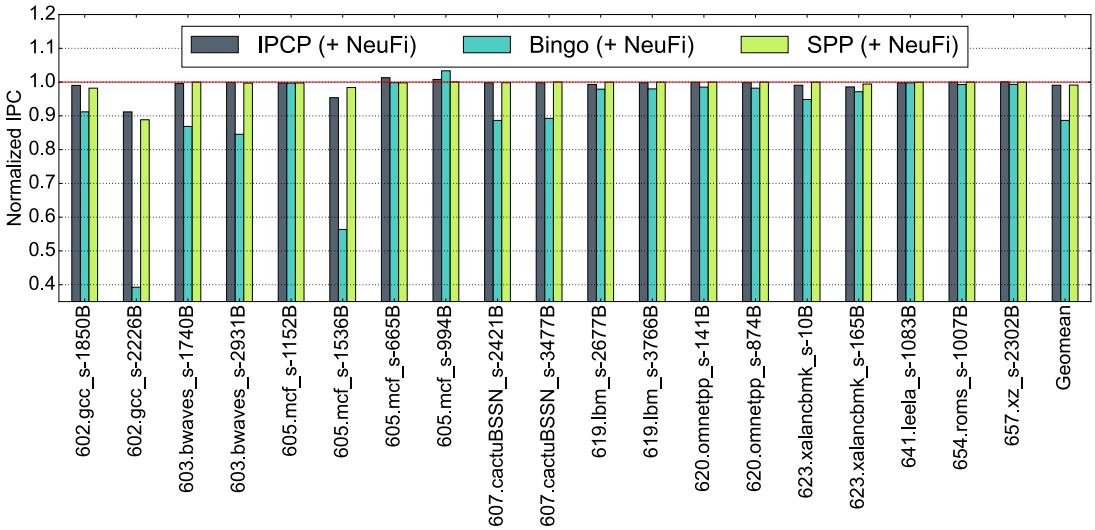


Figure 4.9: Normalized IPCs of NeuFi augmented with various prefetchers on GAP benchmarks. The baselines are the corresponding prefetchers without NeuFi. On a geometric average, the IPC improvement (in %) was **-0.90%** on IPCP [14], **-11.29%** on Bingo [2] and **-0.90%** on SPP [11] prefetcher

Benchmark	**Prefetcher**		
	IPCP	**Bingo**	**SPP**
SPEC2017	0.991	0.887	0.991
GAP	1.011	0.981	0.999

(a) Geometric-mean of the normalized IPCs on SPEC2017 and GAP benchmarks

Benchmark	**Prefetcher**		
	IPCP	**Bingo**	**SPP**
SPEC2017	-0.90 %	-11.29 %	-0.90 %
GAP	+1.10 %	-1.98 %	-0.10 %

(b) IPC improvement (in %) based on the normalized IPCs (from table 4.3a) on SPEC2017 and GAP benchmarks

Table 4.3: Normalized IPCs and IPC improvement with NeuFi

4.5 Prefetch Accuracy

Fig(4.12a) and fig(4.12b) plots the prefetch accuracy of NeuFi augmented with the corresponding prefetchers. Table 4.5 summarizes the results obtained. Prefetch accuracy is

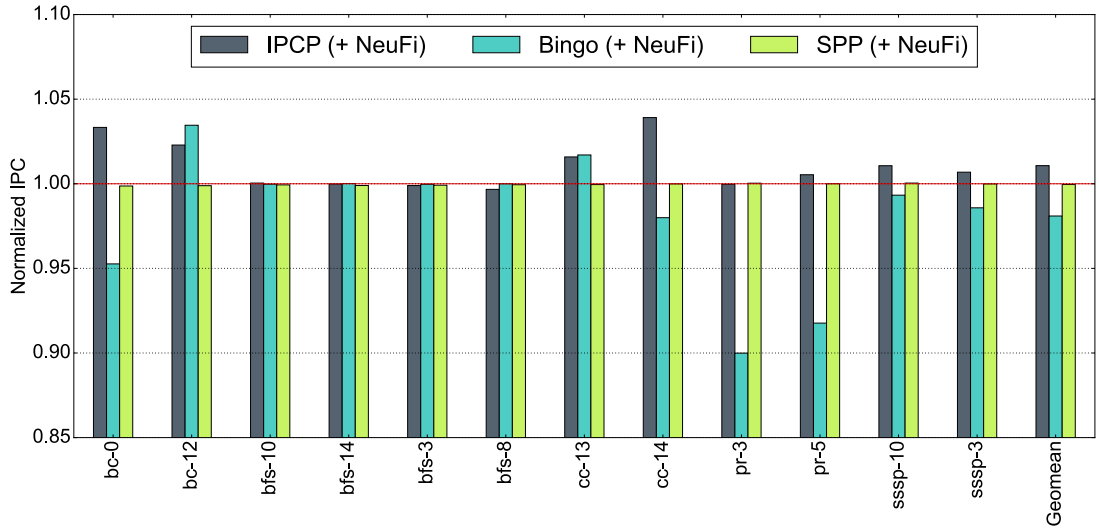


Figure 4.10: Normalized IPCs of NeuFi augmented with various prefetchers on SPEC2017 benchmarks. The baselines are the corresponding prefetchers without NeuFi. On a geometric average, the IPC improvement (in %) was **+1.10%** on IPCP [14], **-1.98%** on Bingo [2] and **-0.10%** on SPP [11] prefetcher

Benchmark	**Prefetcher**		
	IPCP	**Bingo**	**SPP**
SPEC2017	0.960	0.905	0.997
GAP	0.904	0.815	0.999

(a) Normalized accesses to off-chip memory on SPEC2017 and GAP benchmarks

Benchmark	**Prefetcher**		
	IPCP	**Bingo**	**SPP**
SPEC2017	-4.00 %	-9.49 %	-0.30 %
GAP	-9.59 %	-18.50 %	-0.10 %

(b) Reduction in off-chip accesses based on the normalized accesses (from table 4.4a) on SPEC2017 and GAP benchmarks

Table 4.4: Normalized and reduction in off-chip memory accesses

defined as (defined previously in eq(3.1))

$$\text{Prefetch Accuracy} = \frac{\text{No. of Useful Prefetches}}{(\text{No. of Useful} + \text{No. of Useless Prefetches})}$$

It is to be noted that if the *black-box* prefetcher (IPCP [14], Bingo [2] or SPP [11] in this work) decides to issue N prefetch requests, NeuFi (if augmented) would *intercept* those requests and allow only N' ($N' \leq N$) requests to pass. This means that it does not issue any extra request and hence the drop in accuracy as observed in table(4.5), for example, Bingo on GAP benchmarks, is due to blocking potential useful prefetches (reducing the numerator in eq(3.1)) as well as possibly reducing the useless prefetches (reducing the denominator in eq(3.1)).

Prefetcher	**Benchmark**	
	SPEC2017	**GAP**
IPCP	38.333 %	31.027 %
Bingo	68.316 %	74.747 %
SPP	90.034 %	88.661 %
IPCP (+ NeuFi)	38.910 %	33.863 %
Bingo (+ NeuFi)	68.676 %	74.057 %
SPP (+ NeuFi)	90.506 %	88.544 %

Table 4.5: Geometric-mean of the prefetch accuracy on SPEC2017 and GAP benchmarks

4.6 Summary

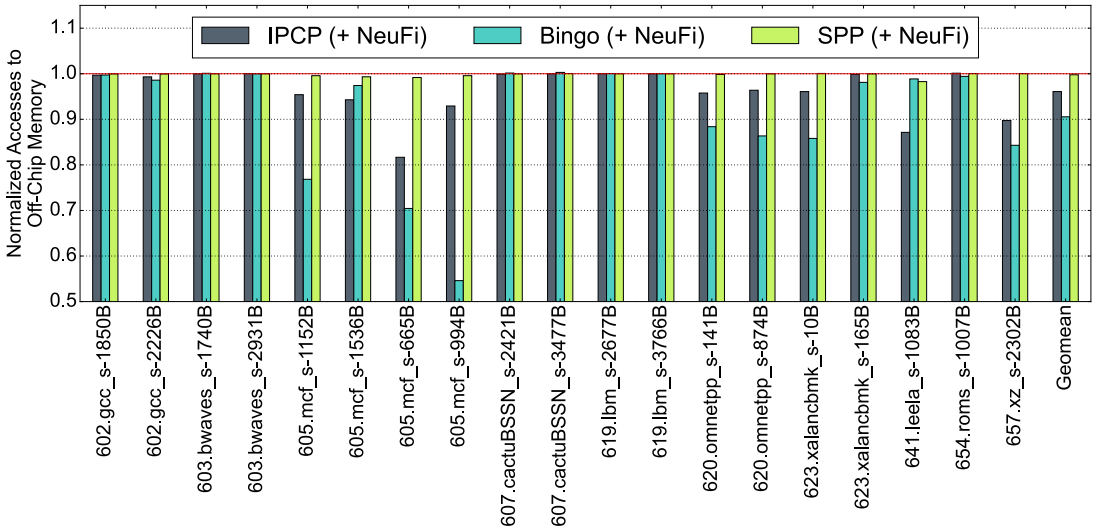
Table-4.6 summarizes the results (of importance) obtained by NeuFi, when augmented with various prefetchers, on SPEC2017 and GAP benchmarks.

Benchmark	**Prefetcher (with NeuFi)**	**IPC Improvement**	**Reduction in Accesses to Off-Chip Memory**	**Accuracy Improvement**
SPEC2017	**IPCP**	-0.924 %	+3.913 %	+1.505 %
	Bingo	-11.345 %	+9.451 %	+0.527 %
	SPP	-0.886 %	+0.220 %	+0.524 %
GAP	**IPCP**	+1.071 %	+9.518 %	+9.410 %
	Bingo	-1.904 %	+18.474 %	-0.923 %
	SPP	-0.044 %	+0.097 %	-0.132 %

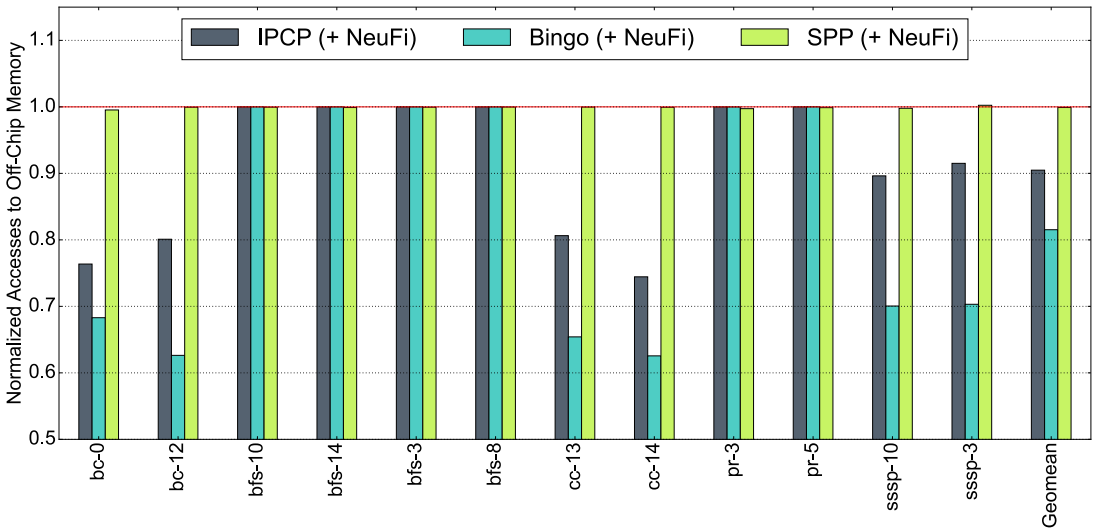
Table 4.6: Summary of the results obtained by NeuFi on SPEC2017 and GAP benchmarks

As can be seen in table(4.6), IPCP [14] obtained the best improvement, both in SPEC2017 and GAP benchmarks. In the case of SPEC2017 benchmarks, although it suffered a performance degradation of **0.924 %**, it reduced the number of accesses to off-chip memory

by **3.913 %**. However, in the case of GAP benchmarks, it not only improved the performance by **1.071 %** but also reduced the number of accesses by a significant **9.518 %**. In the case of Bingo [2], it suffered a significant performance degradation of **11.345 %** in the case of SPEC2017 benchmarks at the cost of **9.451 %** less accesses to off-chip memory. However, for GAP benchmarks, although it suffered a performance degradation of **1.904 %**, it reduced the number of accesses by a significant **18.474 %**. In the case of SPP [11], not much improvement can be seen because the prefetcher is already very accurate, as was shown previously in fig(3.1) and fig(3.2).

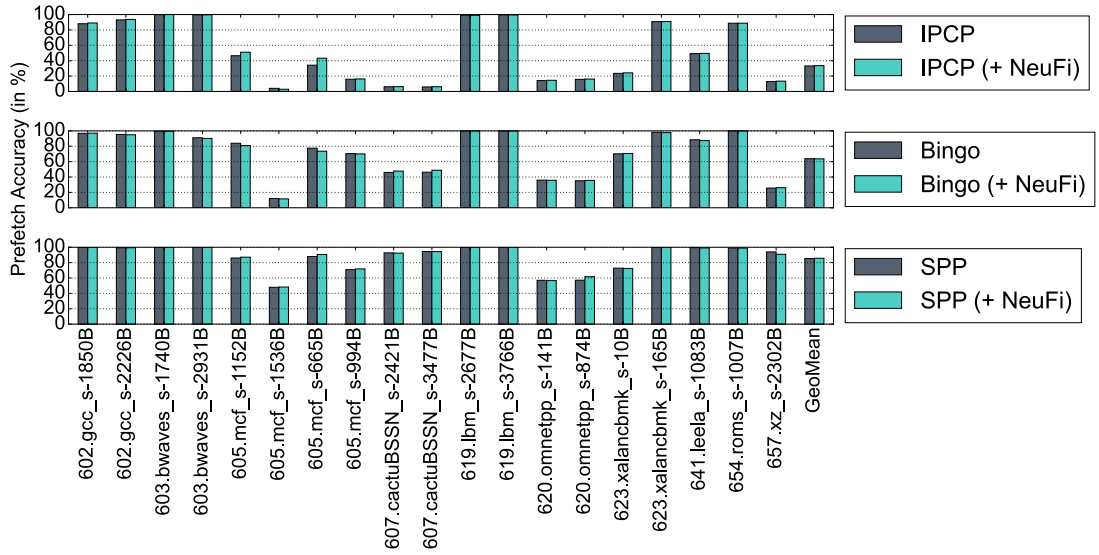


(a) Normalized accesses to off-chip memory by NeuFi augmented with various prefetchers on SPEC2017 benchmarks. The baselines are the corresponding prefetchers without NeuFi. On a geometric average, the reduction (in %) was **4.00%** on IPCP [14], **9.49%** on Bingo [2] and **0.30%** on SPP [11] prefetcher

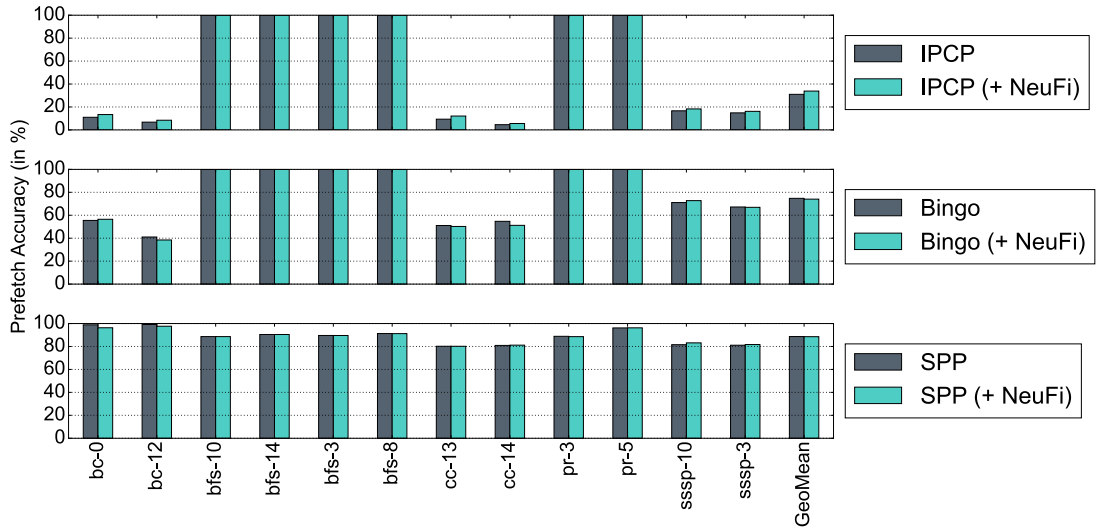


(b) Normalized accesses to off-chip memory by NeuFi augmented with various prefetchers on GAP benchmarks. The baselines are the corresponding prefetchers without NeuFi. On a geometric average, the reduction (in %) was was **9.59%** on IPCP [14], **18.50%** on Bingo [2] and **0.10%** on SPP [11] prefetcher

Figure 4.11: Normalized accesses to off-chip memory by NeuFi augmented with various prefetchers on SPEC2017 and GAP benchmarks



(a) Prefetch accuracy (in %) of various prefetchers with/without NeuFi on SPEC2017 benchmarks. On a geometric average, the accuracy improvement (in %) is **+1.505%** on IPCP [14], **+0.527%** on Bingo [2] and **+0.524%** on SPP [11] prefetcher, based on table(4.5)



(b) Prefetch accuracy (in %) of various prefetchers with/without NeuFi on GAP benchmarks. On a geometric average, the accuracy improvement (in %) is **+9.140%** on IPCP [14], **-0.923%** on Bingo [2] and **-0.132%** on SPP [11] prefetcher, based on table(4.5)

Figure 4.12: Prefetch accuracy (in %) of various prefetchers with/without NeuFi on SPEC2017 and GAP benchmarks.

Chapter 5

Conclusion

Timeliness is one of the core challenges when designing a prefetcher, not considering it leads to the design of sub-optimal prefetchers that are not capable of utilizing their full potential. The first part of this work tried to incorporate timeliness into a prior state-of-the-art (SOTA) (hardware) prefetcher for level-2 caches (L2C), the *Signature Path Prefetcher* (SPP), which does not consider timeliness of its prefetching decisions. The proposed *proof-of-concept* prefetcher, *Reinforced-SPP* (R-SPP) utilized the policy-based *reinforcement learning* framework to update its prefetching policy based on the timeliness of its decisions made in the past and rewarded the actions accordingly. Doing so, led to R-SPP obtaining a performance improvement of **47.056 %** against a baseline system with no prefetching. Whereas SPP (without / with look-ahead) obtained **30.561 %** and **45.673 %** improvement respectively, against the same baseline. The proposed prefetcher in the first part of the work, i.e. R-SPP, offered a performance improvement of **+0.948 %** when compared against a system that has SPP (with look-ahead) as the L2C prefetcher. The main drawback of R-SPP was the prefetch accuracy, which was only **47.743 %** while as SPP (without / with look-ahead) obtained **89.610 %** and **90.462 %** respectively.

Second part of the work focussed on the drawbacks caused due to low accuracy by analyzing two more SOTA prefetchers - IPCP [14] and Bingo [2] under various memory bandwidths and it was emperically observed that the utility of prefetching is highly correlated with the prefetching accuracy. Highly accurate prefetchers which perform relatively lower than less accurate prefetchers, tend to perform much better than the latter, under low memory bandwidth. This lead to modifying the objective to also incorporate improving accuracy alongside timeliness - which led to another proof-of-concept model, Neural-Filter (NeuFi), which is based on supervised deep-learning. NeuFi performed best on GAP benchmarks when augmented with IPCP [14] and Bingo [2]. For IPCP [14], it improved the performance by **+1.071 %** and it reduced the off-chip memory accesses by **+9.518 %**, on a geometric-average. For Bingo [2],it improved the performance by **-1.904 %** (i.e. reduced the performance) but it reduced the off-chip memory accesses by a significant **+18.574 %**, on a geometric-average. It obtained neglible benefit on SPP [11] since it was already accurate enough.

The main drawback of NeuFi currently is that it unlikely to be implementable practically without causing significant impact (negatively) in other areas. Nonetheless, it is a proof-of-concept which had demonstrated its ability to achieve near-similar performance while reducing the number of accesses to off-chip memory by a significant amount - which should motivate designs of similar filters that can be implemented practically, in the future.

Chapter 6

Acknowledgements

I thank my parents for providing me with the necessary resources, financially as well as morally supporting me throughout the entire duration of the course.

I also thank *Prof. Biswabandan Panda* for guiding me throughout the stage-I and stage-II of MTP, taking his precious time every week to go through my weekly progress and providing extremely helpful and critical feedback, which helped me to achieve the results mentioned in this report.

I also thank *Prof. Shivaram Kalyanakrishnan* and *Prof. Preethi Jyothi* for their courses on *Foundations of Intelligent and Learning Agents* (CS-747) and *Foundations of Machine Learning* (CS-725) offered in Autumn 2020, which has let me explore the exciting field of machine learning and which has served as the foundation for this work.

I also thank *Prof. Parag Chaudhuri* for his course on *Communication Skills* (CS-792) offered in Spring 2020, which helped me learn the proper ways to communicate effectively and learn the ethics in scientific research and communication.

Finally, I thank my peers *Veerendrababu Vakkapatla*, all of the CASPER group members, and my fellow peers for helping me and providing me with helpful feedback and insights throughout MTP, and a special mention to *Vishal Pramanik* for helping me with stage-II of the MTP.

Bibliography

[1] Bahdanau, Dzmitry, et al. “Neural Machine Translation by Jointly Learning to Align and Translate.” ArXiv.org, 19 May 2016, https://arxiv.org/abs/1409.0473.

[2] Bakhshalipour, Mohammad, et al. “Bingo Spatial Data Prefetcher.” 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019, https://doi.org/10.1109/hpca.2019.00053.

[3] Bera, Rahul, et al. “DSPatch.” Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, https://doi.org/10.1145/3352460.3358325.

[4] Bera, Rahul, et al. “Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning.” MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture, 2021, https://doi.org/10.1145/3466752.3480114.

[5] Blundell, Charles, et al. “Model-Free Episodic Control.” ArXiv.org, 14 June 2016, https://arxiv.org/abs/1606.04460v1.

[6] Brain, Ashish Vaswani Google, et al. “Attention Is All You Need: Proceedings of the 31st International Conference on Neural Information Processing Systems.” Guide Proceedings, 1 Dec. 2017, https://dl.acm.org/doi/10.5555/3295222.3295349.

[7] Cho, Kyunghyun, et al. “On the Properties of Neural Machine Translation: Encoder-Decoder Approaches.” ArXiv.org, 7 Oct. 2014, https://arxiv.org/abs/1409.1259.

[8] Chung, Junyoung, et al. “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.” ArXiv.org, 11 Dec. 2014, https://arxiv.org/abs/1412.3555.

[9] Heirman, Wim, et al. “Near-Side Prefetch Throttling.” Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, 2018, https://doi.org/10.1145/3243176.3243181.

[10] Hochreiter S., Schmidhuber J”urgen, 1997. Long short-term memory. Neural computation, 9(8), pp.1735–1780.

[11] Kim, Jinchun, et al. “Path Confidence Based Lookahead Prefetching.” 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016, https://doi.org/10.1109/micro.2016.7783763.

[12] Liu, Evan Zheran, et al. “An Imitation Learning Approach for Cache Replacement.” ArXiv.org, 9 July 2020, https://arxiv.org/abs/2006.16239.

[13] Michaud, Pierre. “Best-Offset Hardware Prefetching.” 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, https://doi.org/10.1109/hpca.2016.7446087.

[14] Pakalapati, Samuel, and Biswabandan Panda. “Bouquet of Instruction Pointers: Instruction Pointer Classifier-Based Spatial Hardware Prefetching.” 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, https://doi.org/10.1109/isca45697.2020.00021.

[15] Ros, Alberto. Berti: A per-Page Best-Request-Time Delta Prefetcher. June 2019, https://dpc3.compas.cs.stonybrook.edu/pdfs/Berti.pdf.

[16] Shi, Zhan, et al. “A Hierarchical Neural Model of Data Prefetching.” Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, https://doi.org/10.1145/3445814.3446752.

[17] Wang, Yuxuan, et al. “Tacotron: Towards End-to-End Speech Synthesis.” ArXiv.org, 6 Apr. 2017, https://arxiv.org/abs/1703.10135.