An RL based Prefetcher Design

A Project Report

submitted by

PRASANTH NUNNA (EE10B024)

in partial fulfilment of the requirements

for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2014

THESIS CERTIFICATE

This is to certify that the thesis titled **An RL based Prefetcher Design**, submitted by **Prasanth Nunna**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Shankar Balachandran
Project Guide
Assistant Professor
Dept. of Computer Science
IIT-Madras, 600 036

Dr. Nitin Chandrachoodan
Project Co-Guide
Associate Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 13th June 2014

ACKNOWLEDGEMENTS

I take this opportunity to express my profound gratitude and deep regards to my guide Prof. Shankar Balachandran for his exemplary guidance throughout the course of this project. I gained an excellent exposure to the field of Computer Architecture through this project. My programming skills have also improved to a great extent on account of this project. I thank him for his support during the project in making this happen. I would also like to thank Prof. Nitin Chandrachoodan for being my co-guide and monitoring my performance during the course of this project.

I would like to thank Biswabandan Panda, Anju, Sri Seshan and Pritam for being continuously in help whenever I got stuck. Without their support I could not imagine the outcome of this project. Special thanks to my friends Vageesh, Smit and Vishwanath for helping me in setting up the simulations and interpreting the results. I would like to thank my friends Sasanka, Muni Sreenivas and my wingmates who were very supportive and kept inspiring me during my low times.

Finally and most importantly, I would like to thank my parents and my sister for their moral support without which I would not have been at this position. They always encourage me and make me realize my potential. I am always indebted to their unconditional love.

ABSTRACT

KEYWORDS: Prefetcher ; Exp3 algorithm; Computer Architecture; Reinforcement Learning; High Performance Processor Design.

This report presents the design of a prefetcher using a Reinforcement Learning Algorithm - Exp3. This prefetcher introduces adaptiveness into the conventional prefetchers through the RL algorithm. This adaptiveness is essential for improved performance of the processor. The prefetcher monitors the processor performance by tracking three metrics accuracy, lateness and cache pollution. The prefetcher adjusts its aggressiveness at certain intervals during the program execution. The statistics are collected during the interval and aggressiveness is selected at the of interval, based on the performance of metrics, using exp3 algorithm. The algorithm drives the prefetcher towards optimal state selection to achieve high performance.

The prefetcher is tested for PARSEC benchmark suite and found effective in case of medium granularity workloads. It improved the lateness of benchmarks compared to FDP but performed worse than FDP in case of pollution because of exploration. This prefetcher performs better than No prefetcher and Stride Prefetcher for all of the benchmarks, and is equally good as FDP for most of the benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

FDP	Feedback Directed Prefetcher
Exp3	Exponential Weighted Exploration and Exploitation
RL	Reinforcement Learning
MAB	Multi Arm Bandit
MSHR	Miss Status Holding Register
CMP	Chip Multi Processor
ROI	Region of Interest
IPC	Instructions Per Cycle
FLOPS	FLoating-point Operations Per Second
PARSEC	Princeton Application Repository for Shared-Memory Computers

NOTATION

γ	Parameter to adjust selection between uniform and weighted distribution of bandits
$w_i(t)$	weight of ith bandit at tth instant
$p_i(t)$	probability of selection of ith bandit at tth instant
K	Total number of bandits
$x_j(t)$	current reward of jth bandit at tth instant

CHAPTER 1

INTRODUCTION

Processor speed has increased significantly over the past few decades but memory latency has been a major drawback. The rate of increase in the memory speed is much lesser than the rate at which processor speed is increasing. To compensate this, cache is used between processor and memory. Cache access takes lesser cycles than the memory access. But still cache cannot accommodate all the required data at the same time. This results in cache misses which require access to the memory to get the data. Thus, a cache miss latency is equivalent to a memory access. In order to improve the performance, most useful data must be loaded into the cache. Prefetcher addresses this task by predicting future trends in data access patterns. A prefetcher is attached to a cache and monitors the data entering and leaving it. It loads the most useful data, predicted from aforementioned patterns, into the cache before it is actually accessed by the processor.

But designing a prefetcher with high accuracy is quite challenging because in addition to predicting the future access it should not replace most accessible data from cache. An improper design of prefetcher might result in cache pollution and decrease the performance rather than increasing it. Prefetching aggressively without monitoring might result in decreasing the performance of the processor. The excessive prefetching might replace more useful instructions thereby leading to additional cache misses. Furthermore, it might also lead to usage of bus and memory bandwidth for long time resulting in increasing the latency of each instruction. An adaptive prefetcher monitors the performance and adjusts its prediction patterns. A few adaptive prefetchers have been proposed which adjusts its prefetches based on certain metrics of cache. A feedback directed prefetcher is one such adaptive prefetcher which monitors the performance using Accuracy, Lateness and Pollution of cache. It predicts the future accesses and adjusts prefetch degree based on above parameters. The prefetch degree is the number of blocks to be prefetched at each iteration. But it uses an intuitive and predefined process to identify prefetch degree. This project uses a reinforcement algorithm instead

of an intuitive approach. It uses exp3 algorithm to monitor the performance and predict the trends. This algorithm directs the prefetcher towards the most optimal decision. Thus it is adaptive and directs towards the optimal performance.

The layout of the report is as follows. Chapter 2 gives an overview of various prefetching Techniques, the conventional Stride Prefetcher used in this project and also a brief description on Reinforcement Learning and the Exp3 Algorithm based on which the prefetcher has been designed. Chapter 3 explains the design of the adaptive prefetcher using RL algorithm. It also gives a detailed description of hardware implementation to track the metrics, and employing the exp3 algorithm on the prefetcher problem. Chapter 4 describes the Simulation Environment used for testing the design and the analysis of the results. Chapter 5 concludes the report and presents the future scope of this project.

CHAPTER 2

BACKGROUND

This chapter gives an overview about prefetchers, exp3 algorithm and other information required to understand the project work.

2.1 Prefetching Techniques

Prefetching techniques have been employed in the high performance processors to reduce the gap between processor and memory latency. Caching brings a small amount of data from memory closer to the processor whenever it is required. Prefetching in addition to Caching significantly reduces the gap between memory and processor. Prefetching can be done in different ways like Software based or Hardware based. Software Prefetching relies on compiler technology to selectively insert prefetch instructions. In this method, data will be requested to be prefetched several cycles before the memory access is actually required. But there is some execution overhead due to extra prefetch instructions in software prefetching. Hardware based prefetching requires an additional component attached to the cache. The main advantage through this technique is that prefetches are handled dynamically at run-time unlike software prefetching which requires compiler intervention. But the disadvantage with hardware prefetching is that it requires additional hardware and predicting complex memory access patterns is difficult. Hardware Prefetching in turn can be done in different ways. Popular techniques include stream prefetching, stride prefetching, tagged prefetching etc. In stream prefetching, exclusive buffers are used to allocate stream entries which are trained and tracked to predict access patterns. The stride prefetcher maintains a constant stride by analysing previous accesses and fetches the future data based on the stride that is maintained. In this project, hardware prefetching technique with stride prefetcher has been employed. The following subsection describes the stride prefetching technique in detail that has been used in the project.

2.1.1 Stride Prefetcher

Stride Prefetching is one of the Hardware Prefetching Techniques used to eliminate compulsory misses. In this method, the prefetcher maintains a table of stride entries to keep track of access patterns. An instruction in the entry is associated with a stride which is the difference between the memory location referenced by load instruction and the last address referenced by the load. When the same instruction is accessed again, the block which is at the stride distance from the last address will be prefetched.

The following table illustrates the structure of each entry in the stride table:

Inst. Addr	**Referenced Addr**	**Stride**	**Confidence**
Prog. Counter	block addr	difference	4
...	...	...	...

As shown above each entry in the table has an instruction address, Referenced address, stride entry and confidence. Whenever a prefetcher is notified, it checks if the current instruction is already present in the table. If it is not present a new entry will be created for it with a confidence of zero. If there is a match, the difference between the current address referenced by the instruction and the previous block address from the entry is computed. This is defined to be the new stride. This stride is compared with the existing stride in the table. If the new stride matches with the old stride the confidence of the instruction will be incremented unless it exceeds the maximum confidence limit. Else, the stride will be updated with new stride and the confidence is reset if it is greater than a minimum confidence limit. The referenced address will be updated to the current block address referred by the instruction.

The confidence of each entry is an indicator of the likelihood of access of the blocks at the stride distance from current referenced address. After updating the stride entry, the block at *reference address* + *stride* is requested to be prefetched. The prediction is that the block located at the same offset is more likely to be accessed in the near future. But initiating the prefetch of only the next block might not cover the memory latency to a great extent. Instead multiple blocks located at the same offset can be prefetched at the same time. That is the blocks which are located at the same offset until *reference address* + *N*stride* can be fetched. The **N** is the number of blocks to be prefetched.

This is called as **Prefetch Degree** of the Prefetcher.

Maintaining a constant prefetch degree might be useful for some applications but it might lead to polluting the cache if the degree is too high. Hence, it is highly essential to monitor the effect of prefetching on the performance and vary the degree accordingly. The performance can be monitored by keeping track of certain parameters. These parameters are updated continuously by the prefetcher after certain interval. A few of such parameters which are considered in the development of this project are illustrated in the next subsection.

2.1.2 Performance Parameters

The performance can be tracked by using many different parameters. This project uses three parameters *Prefetch Accuracy, Prefetch Lateness* and *Prefetcher Generated Cache Pollution*. The computation of each of these parameters is clearly illustrated in this subsection

Prefetch Accuracy

Prefetch Accuracy is a measure of how accurately the prefetcher is able to predict the future accesses. It gives an idea of the fraction of prefetched lines accessed by the application before they are replaced. It is defined as the ratio of number of prefetches used by application before they are replaced (*useful prefetches*) to the total number of blocks that are prefetched *i.e.,*

$$Prefetch\ Accuracy = \frac{No.\ of\ Useful\ Prefetches}{Total\ no.of\ prefetches\ issued}$$

The higher the accuracy the better the performance of the processor. Poor accuracy results in more bandwidth occupancy and might also replace useful prefetch lines resulting in more misses.

Prefetch Lateness

Prefetch Lateness is a measure of the timeliness of the prefetch requests. Though the prefetch requests are accurate they are not useful unless they are fetched on time. The prefetch request is said to be *late* if it is still in the queue when a demand request to that block is made by the processor. The prefetch lateness is the ratio of number of late prefetches to the number of prefetches that are accessed by application.

$$Prefetch\ Lateness = \frac{No.\ of\ Late\ Prefetches}{No.\ of\ Useful\ Prefetches}$$

In general lateness can be reduced by increasing the aggressiveness of the prefetcher. The highly aggressive prefetcher issues prefetch requests more often. Hence it has lesser lateness than a conservative prefetcher.

Prefetch Generated Cache Pollution

This parameter is a measure of the pollution created by the prefetcher into the cache. It is highly difficult to compute the exact pollution incurred by the prefetcher, hence an estimate of the pollution is computed. It is defined as the ratio of number of demand misses caused by the prefetcher to the total number of demand misses in the cache.

$$Cache\ Pollution = \frac{No.\ of\ Demand\ Misses\ caused\ by\ Prefetcher}{Total\ no.of\ demand\ misses}$$

The misses caused by prefetcher are the misses which occur when the prefetch requests replace the required blocks. The higher pollution degrades processor performance significantly if not controlled. Besides, it consumes more bandwidth and also replaces useful data resulting in more misses. Hence it is a very crucial parameter to monitor the performance.

The value of these parameters is compared to some thresholds and the degree is updated accordingly. But deciding the optimal degree is slightly tedious task. This task

can be simplified by incorporating a reinforcement algorithm in the prefetcher. This project uses exp3 algorithm to make the decision. This algorithm drives the prefetcher towards optimal degree without much exploration and exploitation. The next section describes the multi arm bandit problem, which is required to understand the algorithm, and the exp3 algorithm in detail.

2.2 Reinforcement Learning Algorithm

Reinforcement Learning is an area of machine learning concerned with how an *agent* ought to take *actions* in an *environment* to maximise cumulative *reward*. Reinforcement Learning allows the machine or software agent to learn its behaviour based on feedback from the environment. This behaviour can be learnt once and for all, or keep on adapting as time goes by. If the problem is modelled with care, some Reinforcement Learning algorithms can converge to the global optimum; this is the ideal behaviour that maximises the reward. As mentioned, there are many different solutions to the problem. The most popular, however, allow the software agent to select an action that will maximise the reward in the long term (and not only in the immediate future).

A basic reinforcement learning model consists of:

1. a set of environment states $\boldsymbol{S}$
2. a set of actions $\boldsymbol{A}$
3. rules of transitioning between states
4. rules that determine *scalar immediate reward* of a transition and
5. rules that describe what agent observes

The agent interacts with the environment in discrete time steps t. At each time step, the agent observes the environment and then an action a_t is chosen to make a transition from state s_t to state s_{t+1}. Now the reward corresponding to the transition (s_t,a_t,s_{t+1}) is determined. The agent repeats this process at each time step so as to maximize total reward from all the transitions. At each step the decision is made based on the previous transitions and the rewards obtained. The agent should consider long term benefits to

achieve optimal reward i.e., though the immediate reward is less sometimes agent may choose that action because of the long term benefits associated with that action.

The following subsections give description about the problem associated with this project and how reinforcement learning algorithm helps to achieve optimal reward for the problem.

2.2.1 Multi-arm bandit problem

Multi arm bandit problem is a sequential resource allocation problem concerning with allocating resources among several alternative projects. These problems are paradigms of fundamental conflict between allocating resources to projects that yield high immediate reward versus allocating resources to projects that yield low immediate reward but yield better results in the long term future. In this MAB problem, resources are allocated to one of the arms which changes its state and thus the player gets a reward corresponding to that transition, and other arms remain in the same state.

This project is related to a specific case of Multi Arm Bandit problem in which the gambler must decide which arm to choose among $\boldsymbol{K}$ non-identical slot machines in a sequence of trails in order to maximize the reward. Initially the gambler has no knowledge about the rewards of each of the arms. Also, the arms do not follow a stochastic process to assign the rewards. The rewards are assigned by an adversary based on the current state of the machine at each trial. Thus this problem is a classical example of the trade off between exploration and exploitation. The exploration being the player trying to find out the best arm and exploitation being the gambler playing the arm believed to give the best trade-off.

The gambler pulls one of the arms at each trial and he gets the pay off pertaining to that bandit at that instant. The goal is to find the arm that gives best pay-off as early as possible and gambling on that bandit to achieve maximum reward. The gambler needs to trade off between exploration and exploitation. If the player plays continuously on the one arm that he thinks is the best one (exploitation) he may fail to discover the actual arm that gives the higher reward. But if he spends more time on exploring the bandits (exploration) to find out the best arm, he may not use the best arm for long time to get

higher reward. Hence, this problem is considered to be a classical example of trade-off between exploration and exploitation. Both should be balanced effectively to get higher reward without consuming more time.

The following subsection explains the exp3 algorithm used to tackle the gambler problem of MAB to attain maximum reward.

2.2.2 Exp3 Algorithm

Exp3 stands for Exponential-weight algorithm for Exploration and Exploitation. As discussed above, the multi-arm bandit problem requires a trade-off between exploration and exploitation to achieve maximum reward. Exp3 algorithm solves this problem by using an exponential weighted factor to decide the bandit.

In this algorithm, each bandit is associated with some weight which are updated at each trial based on the action performed. The action (selection of the arm) is performed based on the weights associated with each arm at that instant. After performing the action, a reward is assigned to the player and all the weights are updated. The updated weights decide which arm to be chosen in the next trial. The weights may be increased or decreased based on whether the reward obtained is good or bad respectively. The psuedo code for the algorithm is shown below and is explained in detail in the following description.

Algorithm 1 Exp3 Algorithm

Parameters: $\gamma \in [0, 1]$
Initialization: $w_i(t) = 1$ for $i = 1, 2, 3, 4....K$

Algorithm:
for each $t = 1, 2, 3, 4, ...$

1: Set $p_i(t) = (1 - \gamma)\frac{w_i(t)}{\sum_{j=1}^{K} w_j(t)} + \frac{\gamma}{K} \quad i = 1, 2, 3, 4, ...K$
2: Select i_t randomly based on $p_1(t), p_2(t), p_3(t), ...p_K(t)$
3: Assign reward $x_{i_t} \in [0, 1]$
4: for $j = 1, 2, 3....K$ set

$$\hat{x}_j(t) = \begin{cases} \frac{x_j(t)}{p_j(t)}, & if\ j = i_t \\ 0, & otherwise \end{cases}$$

$$w_j(t + 1) = w_j(t) exp(\frac{\gamma \hat{x}_j(t)}{K})$$

Initially, all the bandits are assigned a weight of 1. At each instant, the probability

of choosing a bandit is equal to the fraction of the weight of that bandit among total weight. But in Step-1, in addition to this a uniform distribution term $\frac{1}{K}$ is added to the probabilities. This is to ensure that the bandits with very less weights are not left out eventually because they may not not be useful now but might give higher rewards in the future. This term allows the algorithm to try out all the K bandits and gets good estimate of rewards. However the dependence on this term can be controlled by tuning the parameter γ. For $\gamma = 1$ the selection is independent of weights and for $\gamma = 0$ it is independent of uniformity. The user can use a γ close to 1 until a good estimate of rewards is obtained and change it to a value close to 0 after getting the estimate.

After calculating the probabilities, an arm is selected by generating a random number and comparing with the probabilities. Now a reward for this arm which is in the range $[0, 1]$ is selected by the adversary. The adversary may not follow a stochastic process to decide the rewards. The decision may be dependent on the previous bandits selected, current state etc. A reward vector $\hat{x}(t)$ is maintained to modify the weights. After obtaining the rewards this vector is updated as shown in Step-4. The selected arm gets the value $\frac{x_j(t)}{p_j(t)}$ and other arms get zero reward. This compensates the reward of actions that are unlikely to be chosen and guarantees that the expected simulated gain will be proportional to actual gain of the action i.e., the term $\frac{x_j(t)}{p_j(t)}$ increases the chance of selecting the arms with low probability and high reward. The new weights are calculated by multiplying the old weights with the exponential factor as shown in Step-4. The scaling factor $\frac{\gamma}{K}$ in the exponential term ensures that the proportional rewards lie in the range $[0, 1]$.

As it can be seen from the above discussion, the exp3 algorithm trades off between exploration and exploitation through the exponential factor $exp(\frac{\gamma \hat{x}_j(t)}{K})$. Hence it is given the name Exponential weighted algorithm for Exploration and Exploitation. The following chapter explains how the prefetcher problem can be modeled as multi arm bandit problem and how the exp3 algorithm is used in the design of the prefetcher to improve processor performance.

CHAPTER 3

EXP3 PREFETCHER DESIGN

This chapter gives detailed description of the prefetcher design using exp3 algorithm. It explains how prefetcher problem can be modelled as a multi arm bandit problem and how exp3 algorithm can be used to improve the efficiency of the prefetcher.

3.1 Exp3 Prefetcher

Exp3 prefetcher is an improvement on the traditional prefetcher which incorporates adaptiveness in it using exp3 algortihm. In a conventional prefetcher, whenever a cache miss occurs prefetcher is notified of the miss and then the prefetcher anticipates future accesses and issues prefetches based on its type. In the exp3 prefetcher, in addition to that its aggressiveness is varied based on the performance. The duration of process execution is divided into intervals. During each interval, the data required to calculate the parameters is collected and they are updated at the end of each interval. The variations in these parameters during the interval gives an idea of the performance. The aggressiveness is then selected based on whether the current aggressiveness is improving or decreasing the performance. The selection is done using the RL algorithm Exp3. The following sections explain the hardware implementation of calculating the parameters and updating them.

3.2 Calculating the parameters

Three parameters Prefetch Accuracy, Prefetch Lateness and Prefetch generated Cache Pollution have been used in this prefetcher to estimate the performance. These metrics are tracked as discussed below:

3.2.1 Prefetch Accuracy

The prefetch accuracy requires two counters *useful prefetches* and *total prefetches issued*. A *pref* bit is added to each block in the L2 Cache which is useful to keep track of the prefetch requests that are accessed. Whenever a block is inserted into the cache by a prefetch request the *pref* bit is set, and it is reset when a demand request replaces this block. So if a cache hit occurs and the *pref* bit is set then it implies that the prefetched block is accessed, therefore the counters *useful prefetches* and *total prefetches issued* are incremented. If the *pref* bit is not set during a cache hit then it means that the block is not fetched by the prefetcher, hence only *totalprefetches issued* is incremented. The accuracy can be computed by the taking the ratio of the counters *useful prefetches* and *total prefetches issued*.

3.2.2 Prefetch Lateness

The lateness of the prefetch requests is tracked using Miss Status Holding Register (MSHR) queue. MSHR queue keeps track of all the memory requests. A prefetch request is late if the request has an MSHR entry while a demand request to the same block is generated. To keep track of late prefetches the *source* attribute and the *pref-count* bit of MSHR entry are used. Each MSHR entry has a *source* attribute which depicts whether the request is prefetcher generated or a demand request. The *pref-count* bit indicates whether the entry has been counted as late prefetch or not. When a cache miss occurs for a demand request, if the MSHR queue contains a fetch request to the same block that is generated by the prefetcher and its *pref-count* bit is not set, then the late total counter is incremented and *pref-count* bit is set. This counter value is equal to the total number of prefetch requests that are late. The Lateness value can be obtained by taking the ratio of *late total* to *useful prefetches*.

3.2.3 Prefetcher generated Cache Pollution

To keep track of cache pollution it is required to store information about all the blocks dislodged by the prefetcher. But it consumes lot of memory and incurs a heavy overhead on the processor. Hence, it is very difficult to compute cache pollution accurately. But it can be fairly estimated by using a Bloom Filter.

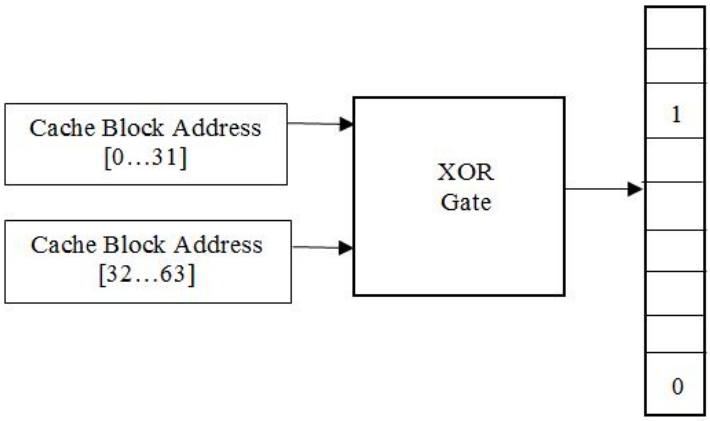

Figure 3.1: Bloom Filter

The bloom filter used in this prefetcher contains an bit vector and a XOR gate. The inputs of the XOR gate are the lower and higher half bits of cache block address as shown in the above figure. The output of the XOR gate corresponds to the address of the bit in the bit vector. When cache block is evicted by a prefetch request the bloom filter is accessed with the block address and the corresponding bit is set in the bit vector. When a demand request replaces a block in the cache then the bit corresponding to that address is reset in the bit vector. Whenever a cache miss occurs, the bloom filter is accessed with the cache block address. If the bit is set, it indicates that the miss is caused by the prefetcher because of the block eviction. The counter *pollution total* is then incremented to keep track of pollution created by the prefetcher. If the bit is reset, then it means that the miss is caused by a demand request. Hence, the counter *cache demand misses*, which keeps track of the misses caused by the demand requests, is incremented. The metric Cache Pollution can be calculated by taking the ratio of the *pollution total* to the *cache demand misses*.

3.2.4 Interval based Counter Update

The program excecution time is divided into intervals and each of the counters mentioned above will be updated at the end of each interval. The interval can be decided by setting a threshold to number of instructions executed or the number of misses in the cache or the number of blocks evicted from the cache etc. In this prefetcher, the number of blocks evicted has been chosen to be the criteria to decide the end of interval. A counter *eviction count* has the value of number of cache blocks evicted in an interval. This counter is incremented whenever a cache block is evicted. When *eviction count* exceeds the threshold $T_{interval}$ then the interval has reached its end. At the end of each interval the above counters are updated by using a feedback based sampling method as shown in the equation below:

$$New\ Counter\ Value = \frac{1}{2}(Counter\ value\ at\ the\ beginning\ of\ the\ interval) + \frac{1}{2}(Count\ during\ the\ interval)$$

This method ensures that the new value takes into account the previous values and the *count during the interval* but the *count during the interval* is given more weight than the previous values. This is to adapt to time varying behaviour of the program. At the end of each interval the *count during the interval* is reset so that it restarts when the next interval begins. After updating all the counter values the metrics Prefetch Accuracy, Prefetch Lateness and Prefetcher generated Cache Pollution are computed to adjust the aggressiveness of the prefetcher.

3.3 Adjusting the Prefetcher aggressiveness using Exp3

This section describes modelling of the prefetcher problem as a multi-arm bandit problem and the use of Exp3 to adjust aggressiveness.

3.3.1 Modelling the prefetcher problem as a multi-arm bandit problem

The real challenge of the prefetcher is to anticipate the future accesses and issue prefetches. To address this, the cache lines can be considered as the arms of the multi-arm bandit problem. But since the number of cache blocks in the L2 cache are very high, the computation of weights for those arms introduces high overhead on the processor. Instead of anticipating the future requests using exp3 it can be done using conventional prefetchers like stride or stream. But the conventional prefetchers will not adapt to the performance of the processor. To tackle this issue, aggressiveness can be adjusted by monitoring the performance. The aggressiveness determines the extent upto which the anticipation can be done by the prefetcher. Hence, the aggressiveness can be selected using the exp3 algorithm. The arms of the prefetcher problem are considered to be various aggressiveness states which prefetcher can attain. Each turn of the player in the multi-arm bandit problem is equivalent to the selection of aggressiveness at the end of each interval. The reward assignment in this case does not follow a specific distribution. Instead, it varies after each interval based on the variation of the parameters computed above. Thus this problem is equivalent to the adversarial multi-arm bandit problem with aggressiveness as arms and performance variation as adversary to decide the reward.

3.3.2 Aggressiveness Selection

Now that the prefetcher problem is modelled as a multi-arm bandit problem exp3 algorithm can be used to make optimum decision at each interval. A constant aggressiveness value throughtout the whole proces might not result in optimal performance because of process behavioural variations. It is necessary to adjust the aggressiveness according to the process after each interval in order to maintain good performance. Since Exp3 algorithm uses Reinforcement Learning, it selects the optimal aggressiveness after each interval.

Aggressiveness State	Prefetch Degree
Very Aggressive	16
Aggressive	8
Middle-of-the-Road	4
Conservative	2
Very Conservative	1

Table 3.1: Aggressiveness States

The aggressiveness states of the prefetcher are divided based on the Prefetch Degree as shown in the table 3.1. Five different states of aggressiveness are considered in this case. These states range from a Prefetch Degree of 1 corresponding to *Very Conservative* state to a Prefetch Degree of 16 corresponding to *Very Aggressive* state. Each of these states is equivalent to each bandit of the multi-arm bandit problem. At the end of each interval i.e., whenever *eviction count* exceeds $T_{interval}$ threshold, the metrics accuray, lateness and pollution are compared to their corresponding thresholds to determine whether each of them is high, medium or low. The accuracy is compared with two different thresholds A_{low} and A_{high} to determine whether the system is at high, medium or low accuracy state. The lateness and pollution are compared to the thresholds $T_{lateness}$ and $T_{pollution}$ respectively to determine whether the system is late and polluting or not.

Case	Prefetch Accuracy	Prefetch Lateness	Cache Pollution	Reward Assignment(State 1, State 2 , State 4, State 8, State 16)
1	High	Late	Not-Polluting	Increasing Reward (0.2,0.4,0.6,0.8,1)
2	High	Late	Polluting	Increasing Reward
3	High	Not-Late	Not-Polluting	Decreasing towards right from current state
4	High	Not-Late	Polluting	Decreasing Reward (1,0.8,0.6,0.4,0.2)
5	Medium	Late	Not-Polluting	Increasing Reward
6	Medium	Late	Polluting	Decreasing Reward
7	Medium	Not-Late	Not-Polluting	Decreasing towards right from current state
8	Medium	Not-Late	Polluting	Decreasing Reward
9	Low	Late	Not-Polluting	Decreasing Reward
10	Low	Late	Polluting	Decreasing Reward
11	Low	Not-Late	Not-Polluting	Decreasing towards left from current state
12	Low	Not-Late	Polluting	Decreasing Reward

Table 3.2: Reward Assignment

After determining the accuracy, lateness and pollution the rewards are assigned to

each bandit as explained in table 3.2. The rewards are divided uniformly for five bandits in the range $[0, 1]$ i.e., every state gets a reward from one of 0.2, 0.4, 0.6, 0.8 and 1. The reward assignment follows four different patterns which are *Increasing Reward, Decreasing Reward, Decreasing Reward from current state towards right* and *Decreasing Reward from current state towards left*. In the *Increasing Reward* pattern, the aggressive state gets the higher reward and vice-versa. This pattern is assigned to the cases which are likely to improve the performance if aggressiveness is increased. In cases 1 and 2 since accuracy is High and the prefetcher is Late increasing the aggressiveness might improve the timeliness of the prefetcher without reducing the accuracy, hence they are assigned with *Increasing Reward*. Similar argument holds for case 5 in which accracy is Medium but the prefetcher is Not-Polluting.

The *Decreasing Reward* is assigned when the prefetcher is creating pollution and consuming bandwidth unnecessarily. It is evident that in cases 4, 8 and 12 there won't be any improvement by increasing the aggressiveness. Since these are Polluting its better to reduce their degree. Thus, the conservative states are assigned higher reward and vice-versa. In cases 6 and 10 since the accuracy is not high and they are polluting they are assigned *Decreasing Reward*. In case 9, though its Not-Polluting it is assigned *Decreasing Reward* pattern to save the bandwidth because the accuracy is Low.

The other two patterns are assigned to cases in which the current state is preferred. Cases 3, 7 and 11 are not-late and not-polluting hence it is better to remain in the same state to keep the benefits of timely prefetches. In *Decreasing Reward from current state towards left* pattern, the conservative states which are closer to current state get higher reward compared to other states. Whereas in *Decreasing Reward from current state towards right* pattern, the aggressive states which are closer to the current state get higher reward. These two patterns are clearly depicted for all the possibilities of the current state in tables 3.3 and 3.4. Case 11 is assigned different pattern from cases 3 and 7, because the accuracy of case 11 is very low and in order to avoid consumption of more bandwidth its better to assign higher reward to conservative ones closer to the current state.

Current State	Reward Assignment(1, 2, 4, 8, 16)
1	(1,0.8,0.6,0.4,0.2)
2	(0.6,1,0.8,0.4,0.2)
4	(0.2,0.6,1,0.8,0.4)
8	(0.2,0.4,0.6,1,0.8)
16	(0.2,0.4,0.6,0.8,1)

Table 3.3: Decreasing reward from center to right

Current State	Reward Assignment(1, 2, 4, 8, 16)
1	(1,0.8,0.6,0.4,0.2)
2	(0.8,1,0.6,0.4,0.2)
4	(0.4,0.8,1,0.6,0.2)
8	(0.2,0.4,0.8,1,0.6)
16	(0.2,0.4,0.6,0.8,1)

Table 3.4: Decreasing reward from center to left

The case is decided by comparing the metrics and the rewards are assigned as per the patterns discussed in table 3.2 after the end of every interval. This assignment serves as the adversary for the prefetcher problem. Now the new state i.e., aggressiveness (Prefetch Degree) is selected by generating a random number and mapping it to the probability distribution of the arms as discussed in 2.2.2. The reward of the selected arm is updated according to the pattern assigned to the current case. The rewards of other states are considered as zero. The new weights are updated according to the Exp3 algorithm as discussed in the earlier sections. Now, until the end of next interval prefetcher operates at the newly assigned degree.

CHAPTER 4

PERFORMANCE STUDY

This chapter deals with the simulation Configuration and the results obtained.

4.1 Simulation Configuration

The simulations are performed using gem5 simulator in full system mode. The prefetcher is attached to the L2 Cache of ALPHA processor. The effectiveness of Exp3 is examined for 4-core CMP. The prefetcher is characterized for multi threaded benchmarks from PARSEC suite. The statistics of each benchmark are collected for `sim-medium` input sets. All the results pertain to the Region of Interest (ROI) of simulation for 500 million instructions. The parameter IPC is used to estimate the performance of the processor. Each core has an L1 i-cache of size 32kB, 2 way and 2 cycle latency, and an L1 d-cache of size 64kB, 2 way and 2 cycle latency. A common L2 Cache of size 512kB, 8 way and 8 cycle latency is connected to all the cores. All the caches have line size of 64B.

The Exp3 Prefetcher is attached to L2 Cache of the processor. Initially the prefetcher is set to *Middle-of-the-Road* Conservative state i.e., Prefetch Degree is initially set to 4. The thresholds A_{high} is set to 0.75, A_{low} to 0.40, $T_{lateness}$ to 0.01 and $T_{pollution}$ is set to 0.005. These thresholds are determined empirically from simulation results. They need not be constant, they can be varied during the process execution time based on processor configuration for improved performance. But these values are considered to be constant during the simulations of this project. The parameter $T_{interval}$ is set to be equal to half the size of cache blocks i.e., 512 for the current configuration of L2 Cache. Whenever the number of evicted blocks exceeds half the size of cache blocks the interval is assumed to end and the counters are updated.

4.2 Analysis of Simulation Results

This section presents the performance of the *Exp3 Prefetcher* compared to three different scenarios-*No Prefetcher, Stride Prefetcher* and *FDP Prefetcher*. The performance is characterized using the metric IPC. Fig 4.1 shows the IPC values of different scenarios when simulated using PARSEC Benchmarks. Figures 4.3, 4.4 and 4.5 shows the average values of the metrics *accuracy, lateness* and *cache pollution* of each of the benchmarks in the ROI for *FDP* and *Exp3 prefetchers*.

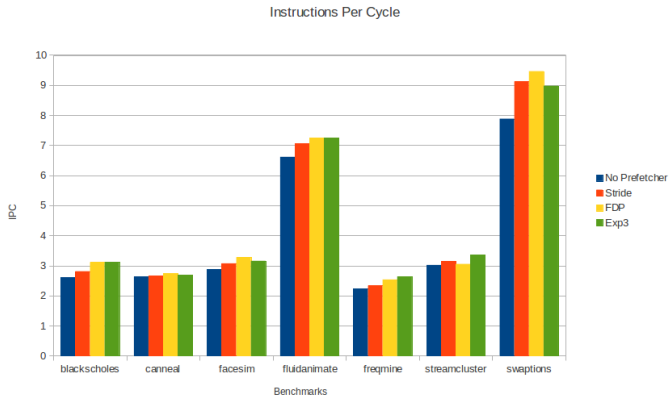

Figure 4.1: Instructions Per Cycle

Clearly *Exp3 Prefetcher* shows significant improvement over *No Prefetcher* and *Stride Prefetcher* for all the benchmarks. It shows an average improvement of 12.01% over *No Prefetcher* and 5.04% over *Stride Prefetcher*. The best performance of *Exp3 Prefetcher* is seen in case of freqmine and streamcluster. The *Exp3 Prefetcher* performs 3.86% and 10.08% better than *FDP Prefetcher* for freqmine and streamcluster respectively. *Exp3* performs significantly worse compared to *FDP Prefetcher* in case of facesim and swaptions. For other benchmarks, *FDP* and *Exp3* are equally good.

Benchmark	Granularity	FLOPS	Locks	Locks/FLOPS
blackscholes	coarse	1.14	0	0
canneal	fine	0.48	34	70.833
facesim	coarse	9.1	14541	1597.91
fluidanimate	fine	2.49	17771909	7137312.85
freqmine	medium	0.0	990025	inf
streamcluster	medium	11.6	191	16.466
swaptions	coarse	2.62	23	8.778

Table 4.1: Characterization of Benchmarks

Table 4.1 shows the number of FLOPS and locks associated with each benchmark.

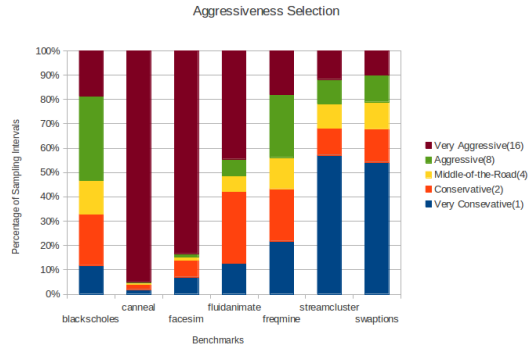

Figure 4.2: Aggressiveness Allocation

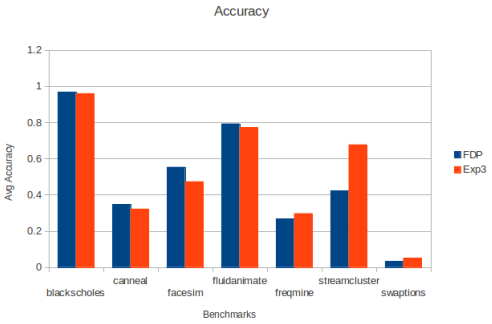

Figure 4.3: Average Accuracy

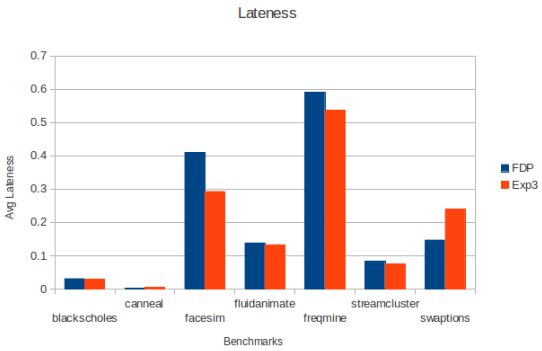

Figure 4.4: Average Lateness

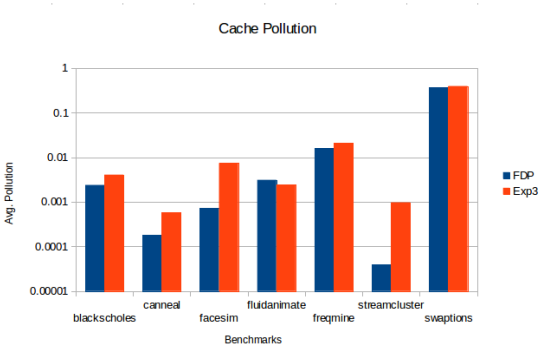

Figure 4.5: Average Pollution

Fig 4.2 shows the percentage of time during which each state is selected for each benchmark. The third column of table 4.1 indicates the number of locks per FLOPS. It is evident that higher the value of this ratio higher the proportion of time spent in highly aggressive state and vice-versa. Canneal, Facesim and Fluidanimate remain in aggressive states for longer duration whereas streamcluser and swaptions whose ratio is less remain in conservative states for longer duration. As the locks per operation increases there will be more waiting time due to synchronization. Hence large amount of cache bandwidth is available. This non-utilization of cache bandwidth results in increased lateness and less pollution. Hence the algorithm assigns very aggressive states to these benchmarks for longer duration.

Unfortunately the *most aggressive* and *very conservative* states (except streamcluster because it is barrier based implemention) shows reduction in the performance. For *most aggressive* ones, the excessive prefetching for long time evicts the required blocks from the cache increasing the cache pollution thus resulting in performance degradation. These trends for canneal and facesim are evident from the Fig 4.5. In case of

swaptions, the conservative approach for long time increases the chances of miss in the cache, thereby leading to more requests in the demand queue which in turn increases the lateness of prefetch requests as observed in Fig 4.4. This increase in the lateness attributes the performance degradation. The Conservative prefetching also avoids the unnecessary evictions from the cache thereby increasing the accuracy without affecting the pollution, hence the accuracy of the swaptions is high though the performance is less.

The selected aggressiveness of freqmine and streamcluster is sufficient to incorporate the required blocks into the cache as these are of *medium* granularity.The *medium* granularity would favour better performance as they require low memory bandwidth at the caches. This explains the better performance of the *medium* granularity benchmarks freqmine and streamcluster.

The lateness of *most aggressive* and *middle-of-the-road* benchmarks is better as the cache lines are burdened with the prefetch traffic and reduce the tendency of the prefetched data being still backlogged in the buffer.

The *most aggressive* benchmarks like canneal and facesim have shown high pollution as expected but the *middle-of-the-road* conservative ones also perform worse than *FDP* in terms of pollution. This is because *Exp3* initially explores through all the possible states, sometimes allowing the low probability ones as well which might result in evicting the required blocks thereby increasing the pollution.

The trends of accuracy is similar to that of IPC. This is because IPC is a cumulative effect of the three metrics. Ideally high accuracy, low lateness and low pollution should result in high IPC. But since the trends of Lateness and Pollution is same across all the other benchmarks, accuracy is the only deciding factor for IPC. This explains the similar trends of accuracy and IPC. The Exp3 Prefetcher is found to improve lateness of almost all the benchmarks but creates pollution overhead because of exploration. It is aggressive towards high locks/FLOPS ratioed benchmarks and shows improved performance for *medium* granularity benchmarks.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

This report presented a prefetcher based on Reinforcement Learning Algorithm- Exp3. This adaptive prefetcher takes into account of the long term benefits unlike other adaptive prefetchers which uses limited horizon stochastic control design. The reinforcement learning approach drives the prefetcher towards optimal decision making to achieve high performance.

The proposed prefetcher shows high performance for workloads with medium granularity. This prefetcher proves effective to improve the lateness for almost all of the workloads. But because of the exploration, it adds a little overhead in terms of the pollution. IPC follows the accuracy trends, as lateness and pollution trends are almost same across all the benchmarks.

5.2 Future Work

A few changes can be done on the above prefetcher to improve its performance. A dynamic cache insertion policy can be designed instead of conventional LRU policy to reduce the pollution incorporated by the prefetcher and also to improve the performance. In addition to accuracy, lateness and pollution other metrics like prefetcher coverage can also be included to get better results. The thresholds of the metrics can also be dynamically adjusted instead of having deterministic values. The γ value in the Exp3 algorithm can also be dynamically controlled to choose between uniform selection and weight based selection of states for improvement in the results. This model can be adopted and tested on other conventional prefetchers like stream and tagged prefetchers.

REFERENCES

[1] **S. Srinath** *et al.* "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers". In *HPCA-13*, 2007.

[2] **P. G. Emma, A. Hartstein, T. R. Puzak** and **V. Srinivasan**. "Exploring the limits of prefetching".*IBM J. RES. & DEV.* VOL. 49 NO. 1 JANUARY 2005

[3] **C. Bienia** *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications", in *PACT 2008*, pp.72-81.

[4] **Peter Auer** *et al.*,"Gambling in a rigged casino: The adverserial multi-arm bandit problem", *IEEE 1995*.

[5] **Peter Auer** *et al.*,"The non-stochastic multi-arm bandit problem", *36th Annual Symposium on Foundations of Computer Science*, pp.322-331,1995.

[6] **Engin İpek, Onur Mutlu, José F. Martínez** and **Rich Caruana**. "Self Optimizing Memory Controllers: A Reinforcement Learning Approach", in *ISCA*, June 2008.

[7] **Biswabandan Panda** and **Shankar Balachandran**. "CSHARP:Coherence and SHaring Aware Replacement Policies for Parallel Applications". *IEEE 24th International Symposium on Computer Architecture and High Performance Computing 2012*.