
Identifying and Prefetching Performance-critical Loads

A thesis submitted in fulfillment of the requirements

for the degree of Master of Technology

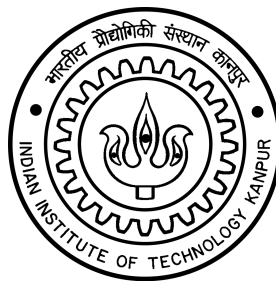
by

Jatin Dev

18111027

under the guidance of

Dr. Mainak Chaudhuri



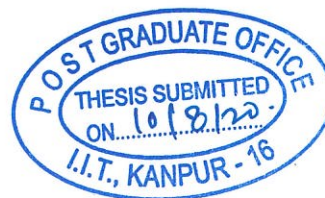
to the

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

Aug, 2020

CERTIFICATE



It is certified that the work contained in the thesis titled “**Identifying and Prefetching Performance-critical Loads**” by “**Jatin Dev**” has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

A handwritten signature in black ink that reads "Mainak Chaudhuri".

(Signature)

Dr. Mainak Chaudhuri

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Kanpur

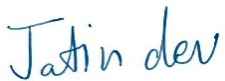
Kanpur, 208016

Aug, 2020

Statement of Thesis Preparation

I, Jatin Dev, declare that this thesis titled, **“Identifying and Prefetching Performance-critical Loads”**, hereby submitted in partial fulfillment of the requirements for the degree of Master of Technology and the work contained here in are my own. I further confirm that:

1. The “Thesis Guide” was referred to for preparing the thesis.
2. Specifications regarding thesis format have been closely followed.
3. The contents of the thesis have been organized based on the guidelines.
4. The thesis has been prepared without resorting to plagiarism.
5. All sources used have been cited appropriately.
6. The thesis has not been submitted elsewhere for a degree.



(Signature)

Name: Jatin Dev

Roll.No.: 18111027

Department of Computer Science and Engineering

Aug, 2020

ABSTRACT

Prefetching is an important speculative technique that relies on prediction of the sequence of addresses that the processor will access in near-future. A prefetcher, based on this prediction, fetches the contents of the predicted addresses and places them in the processor’s cache hierarchy. Over time, prefetchers have become more and more complex with some of the latest prefetching proposals advocating the use of machine learning. The primary goal of a prefetcher is to hide the latency of the load instructions. Although most prefetchers treat all load instructions equally, some load instructions are known to be more performance-critical and introduce bigger latency on the critical path of execution compared to the rest of the loads. The criticality of a load can be dependent on various factors like cycles it takes to fetch it, number of instructions that are dependent on it, etc.. In this thesis, we set out to identify the static load instructions that turn out to be performance-critical during execution and use this information in prefetching. There are two distinct phases in the proposed scheme. First, we identify and store the critical static loads at run-time. Second, we incorporate this information in an existing prefetcher in various different ways to address issues like cache pollution and memory bandwidth consumption. Our simulation-based study focuses on a contemporary three-level cache hierarchy model with the third level (L3) being shared by multiple on-chip cores and the first two levels (L1 and L2) being private to each core. Each core’s L2 cache is equipped with the signature path-based prefetcher (SPP). We use the set of critical loads to optimize the prefetch lookahead, prefetch degree, and other parameters of SPP. In a single-core setting, we are able to achieve an average 1% performance improvement compared to SPP for a large subset of the SPEC CPU 2017 workload traces. Also, a low bandwidth variant of our proposal can maintain the performance of SPP while saving memory bandwidth. On multi-core configurations, we evaluate different flavors of our proposal geared toward different levels of memory bandwidth consumption and find that the low bandwidth flavor can maintain the performance of SPP while saving bandwidth. Across all configurations, we observe that the use of critical load addresses for triggering prefetches significantly reduces the number of times the prefetcher is activated. This is expected to lower the dynamic energy expended in the prefetcher.

Acknowledgements

I express my sincere and deepest gratitude to my supervisor, Dr.Mainak Chaudhuri. I'm always thankful to him for giving me the freedom to learn and explore in my area of interest.

Jatin Dev

Contents

Acknowledgements	vi
List of Figures	ix
1 Introduction	1
1.1 An Overview of a Typical Hardware Prefetcher	1
1.2 An Overview of Delinquent/Critical Loads	3
1.3 Brief Problem Statement	4
1.4 Outline of Our Approach	4
1.5 Organization of the Thesis	5
2 Related Work	6
2.1 Critical Loads	6
2.2 Data Prefetching	6
2.3 SPP	7
2.4 Prefetch Filtering	9
3 ROB Stall and Load Dependents	11
3.1 ROB Stall	11
3.2 Dependent Instructions	11
3.3 Computing ROB Delay and Dependencies at Run-time	12
3.4 Storage of Critical IPs	19
3.5 Determining Critical IPs	20
4 Computing Prefetch Lookahead for Critical Loads	22
4.1 IP Accuracy-based Prefetching	22
4.2 Using SPP Lookahead Predictor	23
4.3 Prediction using Global Prefetch Accuracy	24
4.4 Deciding Fill Level of Prefetches	25

4.5	Periodic Updates of Parameters and Initialization	25
4.6	Dynamic Thresholds	25
4.7	Depth-wise and Total Prefetch-wise Approaches	26
5	Experiments and Results	28
5.1	Single Core Results	29
5.2	High Bandwidth Proposal with Different Depths	34
5.3	Quad-core Homogeneous Results	37
5.4	Quad-Core Heterogeneous Workloads	40
6	Conclusions and Future Work	44

List of Figures

1.1	Typical L2C prefetcher	2
2.1	Global history buffer [9]	7
2.2	SPP operations [1]	8
2.3	SPP delta and pattern table updates [1]	9
3.1	Experimental results for ROB delay and dependency (spread over the next six pages)	12
3.2	Structures to capture and use critical load IPs in a typical CPU	19
4.1	SPP with low bandwidth consumption	24
4.2	SPP with depth prediction configuration	27
5.1	Performance comparison for single-core configuration	29
5.2	Number of prefetch lookups at L2C for single-core configuration	30
5.3	Number of prefetch lookups at LLC for single-core configuration	31
5.4	DRAM prefetch request count for single core configuration	31
5.5	Number of load addresses passed to SPP for single core configuration	32
5.6	Accuracy of prefetches at L2C and LLC for single core configuration	33
5.7	Analysis of high bandwidth proposal's effect on IPC for single core configuration .	33
5.8	Depth vs. IPC for single core configuration	34
5.9	Depth vs. L2C prefetch count for single core configuration	35
5.10	Depth vs. LLC prefetch count for single core configuration	35
5.11	Depth vs. DRAM prefetch count for single core configuration	36
5.12	Depth vs. number of addresses input to the prefetcher for single core configuration	36
5.13	Performance comparison for quad-core homogeneous workloads	37
5.14	Number of L2C prefetches for quad-core homogeneous workloads	38
5.15	Number of LLC prefetches for quad-core homogeneous workloads	38
5.16	Number of DRAM prefetches for quad-core homogeneous workloads	39
5.17	Accuracy comparison for quad-core homogeneous workloads	39

5.18	Number of addresses input to the prefetcher for quad-core homogeneous workloads	40
5.19	Performance comparison for quad-core hetroogeneous workloads	41
5.20	L2C prefetch count comparison for quad-core heterogeneous workloads	41
5.21	LLC prefetch count comparison for quad-core heterogeneous workloads	42
5.22	DRAM prefetch count comparison for quad-core heterogeneous workloads	42
5.23	Accuracy comparison for quad-core heterogeneous workloads	43
5.24	Number of addresses input to the prefetcher for quad-core heterogeneous workloads	43

Chapter 1

Introduction

Prefetching instruction and data has taken a central position in processor architecture. This thesis squarely focuses on data prefetching. Data prefetching can be carried out in two possible ways, namely through hardware prediction of access stream and through the assistance of compiler-inserted prefetch instructions guided by static analysis or profile passes. While the former is known as hardware prefetching, the latter is known as software prefetching. This thesis deals with a certain class of hardware data prefetching. In this chapter, we first present an overview of a typical hardware data prefetcher. Next, we briefly introduce the concept of delinquent or critical loads leading to the problem statement of the thesis and our approach to the problem. We conclude this chapter with an outline of the thesis.

1.1 An Overview of a Typical Hardware Prefetcher

Making predictions about future behaviour of a program is at the core of many of the microarchitecture techniques like cache replacement policies, branch predictors, etc.. Data prefetching is also one such prediction technique that fetches data required in advance by learning the past memory access behaviour of a program. Data prefetchers predict and fetch addresses of the access stream into the cache hierarchy. The position where a data prefetcher sits in the cache hierarchy can differ from one design to another. In this thesis, we will consider a simple design where a data prefetcher is attached to the last-level of the private cache hierarchy of each core.

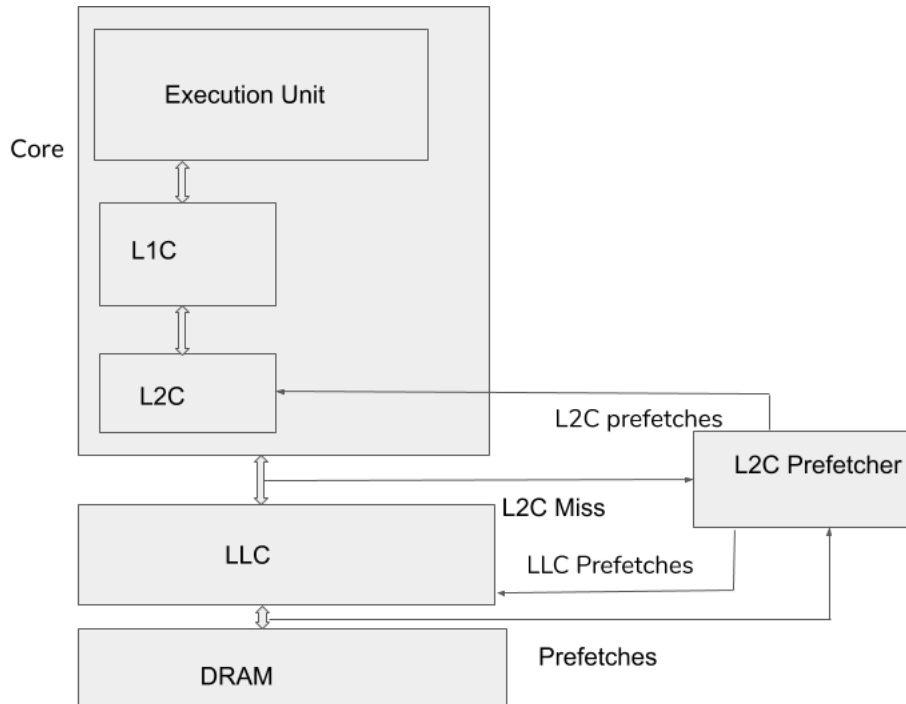


Figure 1.1: Typical L2C prefetcher

There are many types of data prefetchers like next-line, stream buffers-based, history-based, etc.. Each one uses a different technique to predict the address stream of the program data. At a high level, two broad categories of data prefetchers exist. One category that attempts to discover patterns in the data address stream from the control flow of the program uses the instruction pointers (IPs) of static load instructions for tagging prefetch patterns. These IP-based prefetchers discover the data address pattern for each static load instruction associated with an IP. One popular pattern that a prefetcher looks for is the delta between two consecutive load addresses sourced by the same static load instruction at an IP. This pattern is also known as the stride pattern. The other category of prefetchers exploits data flow of the program and uses the data addresses for tagging prefetch patterns. The stride or delta pattern is also popular among this class of prefetchers. There are also prefetchers that have used a combination of IPs and data addresses to discover and tag prefetch patterns.

A prefetch request can be triggered at numerous events. One commonly used trigger event is a cache miss. The prefetch requests for a sequence of addresses are sent out following the request for the address that has missed in the cache. Similarly, at other events, a prediction can be made and the predicted address sequence can be sent as prefetch requests.

Figure 1.1 describes the basic operations of a typical hardware prefetcher attached to the private L2 cache (L2C). Generally, the L2C prefetcher prefetches all data blocks up to L2C only. Intelligent prefetchers can filter out less important prefetches and put them only in the larger last-

level cache (LLC), while more critical blocks are sent further into the cache hierarchy up to the L2C. Basic types of prefetchers just prefetch one or some of the next few blocks for a missed address with a fixed or a dynamic stride. The number of prefetch requests sent out together is usually referred to as the degree or depth of the prefetcher. Smart prefetchers adaptively modulate the depth. They can also select between a set of stride or delta values based on a dynamically learned confidence.

Various metrics and terminology used in prefetching literature are introduced below.

- **Timeliness:** A block should be ideally prefetched just in time synchronized with a demand access to the block from the CPU. An early prefetched block has a high chance of getting replaced from the cache before getting accessed. A late prefetched block fails to hide the latency of a miss completely. However, depending on how late a prefetch is a fraction of the miss latency can be saved.
- **Accuracy:** The accuracy of a prefetcher is the ratio of the number of prefetched blocks that are used to the total number of prefetched blocks.
- **Coverage:** Coverage of a prefetcher is the ratio of the total number of demand misses that are eliminated by the prefetcher to the total number of demand misses generated without using the prefetcher.
- **Lookahead or Degree or Depth:** Lookahead or degree or depth is the number of prefetch requests generated together whenever the prefetcher is triggered.
- **Cache pollution:** Unwanted or less important cache blocks that gets prefetched into the cache can replace useful blocks and pollute the cache degrading the performance.

1.2 An Overview of Delinquent/Critical Loads

Certain static loads or certain memory addresses may be more important for performance than others. These static load IPs or data addresses should be given more importance while prefetching. Previous works have shown that there are only a small number of addresses that are responsible for most of the memory stalls arising from cache misses. It has been shown that out of all commit stalls, 60% of the stalls are caused by a small set of loads [6]. These loads are called Loads Incurring Majority of Commit Stalls (LIMCOS). Subsequent studies further classified these loads and termed the more important loads as delinquent loads [4]. These loads are responsible for most of the cache misses. It was found that ten or fewer static loads cause more than 80% of L1 data cache misses. In this thesis, we use two measures of criticality of static loads: (i) the time for which the re-order

buffer (ROB) head remains blocked by a load instruction waiting to be retired and (ii) the number of instructions that are dependent on a load instruction.

1.3 Brief Problem Statement

Most of the prefetchers don't use criticality information while learning prefetchable patterns or prefetching. However, prefetching only critical loads may result in similar or better performance with less consumption of bandwidth. The main objective of the thesis is to use criticality information to extract important IPs corresponding to the set of performance-critical static loads and use them in tuning prefetcher parameters. We evaluate our proposal in terms of performance, bandwidth consumption, and other metrics. Achieving these objectives involves various challenges discussed below.

- Deciding criticality threshold values for ROB head block time and count of dependents on a load or designing any other mechanism to effectively compute them at run-time.
- Using this criticality information in the underlying prefetcher and making changes to the prefetching algorithm to use this information effectively.
- Reducing cache pollution while maintaining performance with less bandwidth.

1.4 Outline of Our Approach

We dynamically determine the set of critical static load IPs using the criticality metrics outlined above (i.e., ROB head stall and dependence count) and use only those IPs for prefetch injection. To obtain the optimal values of thresholds related to ROB head stall and dependencies, we use an empirical approach.

We use the signature path-based prefetcher (SPP) as the underlying baseline prefetcher (this prefetcher is discussed in detail in the next chapter). We predict lookahead depth for critical IPs using our own techniques instead of the in-built mechanism of SPP. We predict the lookahead depth using parameters such as prefetch accuracy of a static load sourced by an IP and prefetch accuracy at the L2C and LLC with additional techniques that help effectively measure these parameters. We aim at maintaining or improving SPP performance while consuming less bandwidth and other resources. To meet this goal, we have designed two separate techniques that change the underlying prefetcher in different ways. One design is less aggressive and consumes less bandwidth while maintaining the performance of SPP. The other design is more aggressive resulting in higher bandwidth consumption and better performance compared to the underlying prefetcher. We have

conducted extensive empirical evaluation to reach the optimal parameter setting and demonstrated the effects of prefetching based on critical IPs in various different scenarios.

1.5 Organization of the Thesis

- Chapter 2 discusses the related work on critical loads and prefetching that we make use of in this thesis.
- Chapter 3 discusses in detail the ROB head stall and dependence counts and the techniques we use to measure and record them at run-time.
- Chapter 4 discusses various techniques that we use to optimally decide the various threshold values and to efficiently measure and use various parameters like IP-based prefetch accuracy at L2C and LLC. It also discusses the different techniques that we design for low and high bandwidth settings.
- Chapter 5 demonstrates and discusses single-core and multi-core simulation results.
- Chapter 6 concludes the thesis and points to some ideas for the future work.

Chapter 2

Related Work

In the following, we discuss the relevant pieces of work on critical memory accesses, data prefetching, and prefetch filtering.

2.1 Critical Loads

Previous studies have shown that only a small number of static loads is responsible for most of the memory stall. Specifically, it has been shown that out of all commit stalls, 60% are caused by loads [6]. These loads are referred to as Loads Incurring Majority of Commit Stalls (LIMCOS). The authors of this work introduce Focused Prefetching that uses classifiers to filter the training stream seen by the prefetcher i.e., only the misses suffered by the LIMCOS identified by the classifier act as the training stream for the prefetcher. By focusing on misses suffered by LIMCOS, it allows the prefetcher to eliminate misses that have a significant impact on performance. This technique can be used with any prefetcher. Subsequent studies have classified these loads further and termed the more important loads as the delinquent loads [4]. These loads are responsible for most of the cache misses. This study found that ten or fewer static loads cause more than 80% of L1 data cache misses. The solution proposed by this study precomputes the delinquent load addresses in a separate thread and injects prefetches. By limiting prefetches to the delinquent loads only, the proposal reduces resource contention in fetch, execute, and memory system bandwidth.

2.2 Data Prefetching

Data prefetching is an important component of a processor which speculates and fetches data blocks into the cache hierarchy before they are actually used. It is impossible to do justice to the vast literature on data prefetching in a section of a thesis. In the following, we present the general prefetching technique and then discuss in detail the SPP which we use as the baseline prefetcher.

There are different types of data prefetchers differing primarily in terms of the algorithm to discover prefetch patterns and the depth of prefetching. Some of the early prefetchers prefetch only the line following the missed line. Stream prefetchers prefetch multiple next lines and stride prefetchers prefetch the next lines in a sequence that form an arithmetic progression of block addresses with common difference d . The global history buffer (GHB)-based prefetchers can implement any history-based prefetching technique [9]. Figure 2.1 shows a GHB-based prefetcher.

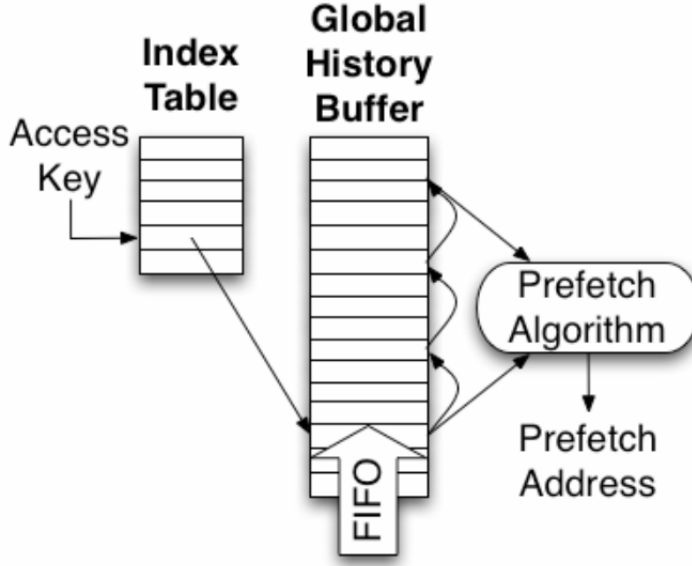


Figure 2.1: Global history buffer [9]

The GHB holds the n most recent L2 cache miss addresses. Each GHB entry stores a global miss address and a link pointer. The link pointers are used to chain the GHB entries that form a time-ordered sequence of addresses having the same index table key. Depending on the key that is used for indexing table, any history-based prefetch method can be realized.

2.3 SPP

SPP can predict multiple delta or stride values and complex strides across physical page boundaries [1]. It stores the memory access patterns in a compressed form called signatures. It also uses a confidence-based mechanism to throttle the aggressiveness of the prefetcher. It uses a combination of the previous delta values for each page as a signature to index into a pattern table which gives the next delta and its confidence. SPP uses a signature table to store the signatures and a pattern table for storing the delta values and their corresponding confidences (Figure 2.2).

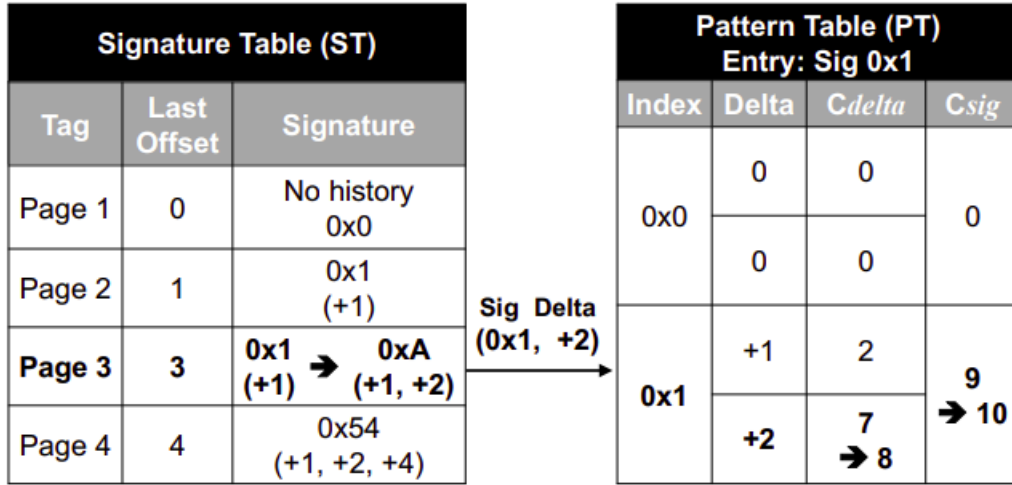
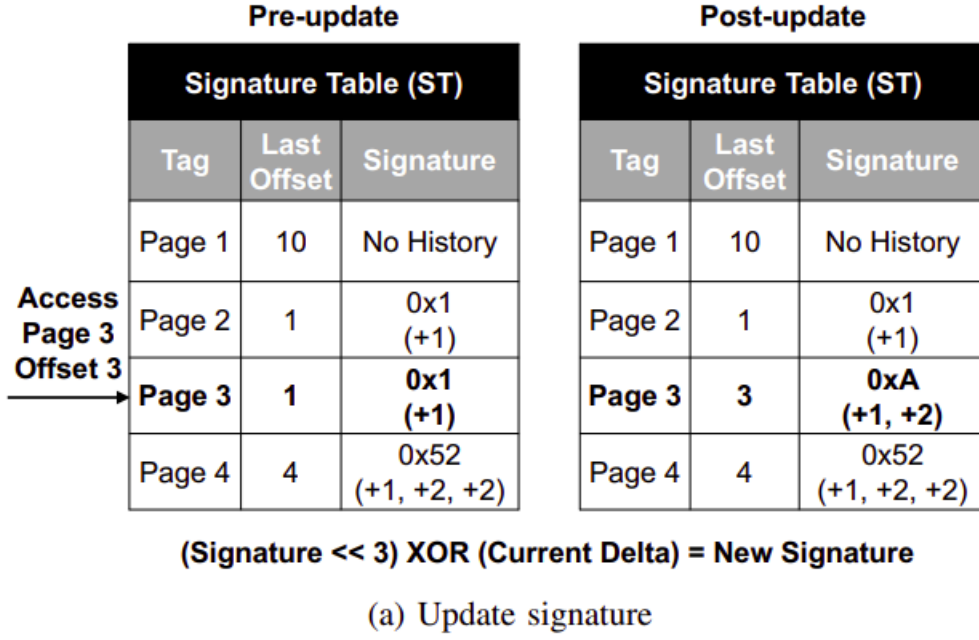


Figure 2.2: SPP operations [1]

Delta is used to generate the new signature, which is used to again access the pattern table to generate lookahead prefetches until the path confidence falls below a threshold (Figure 2.3). By using this mechanism SPP can learn complex access streams.

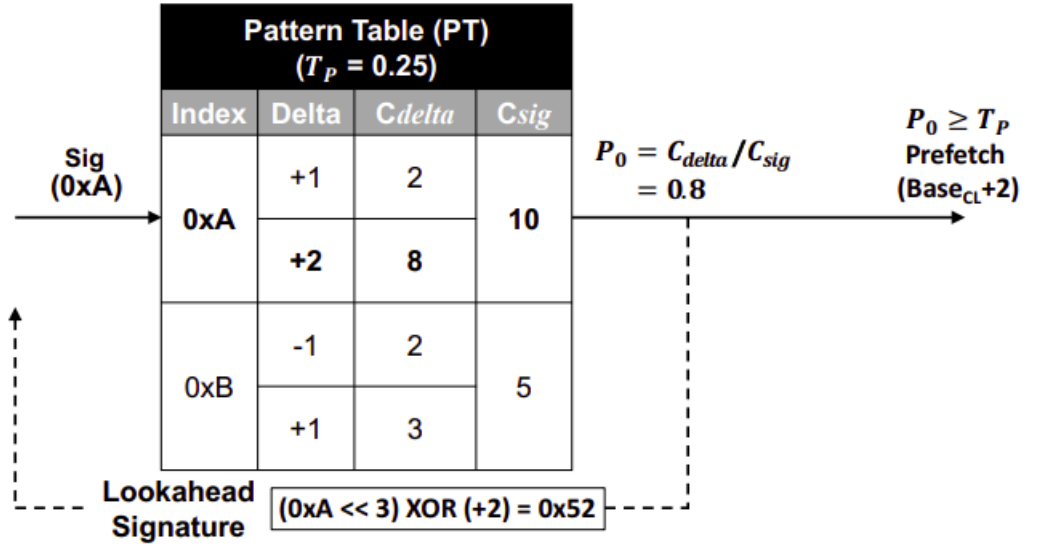
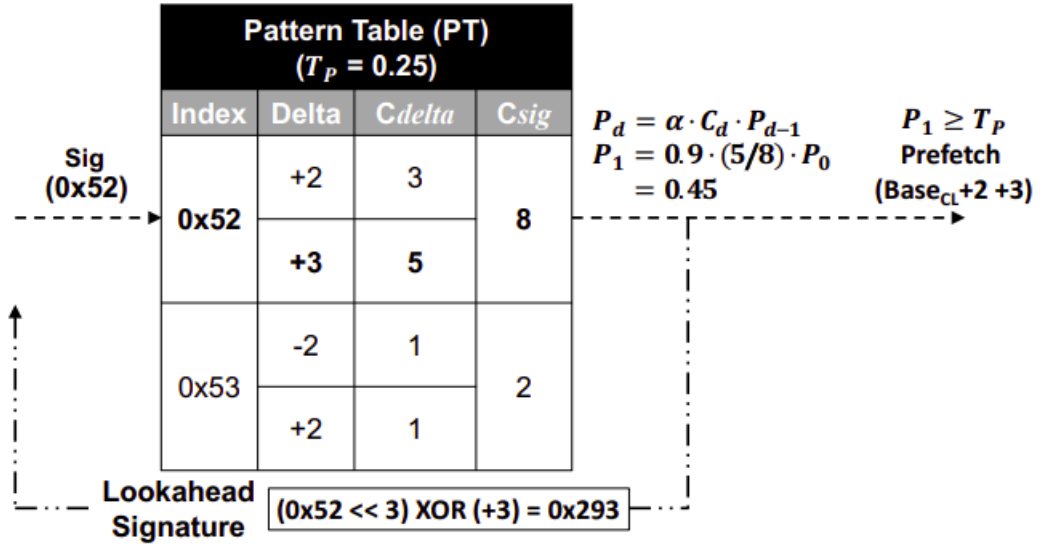
(a) Prefetch (Lookahead depth $d=0$)(b) Lookahead prefetch (Lookahead depth $d=1$)

Figure 2.3: SPP delta and pattern table updates [1]

2.4 Prefetch Filtering

A very high degree of prefetching can result in cache pollution where useful cache blocks are replaced by unnecessarily early prefetches. On the other hand, a very low degree may not be able to take advantage of prefetching and may result in low coverage. There are types of workloads and phases of execution of the same workload which need prefetches with higher depth, while other workloads or phases may not require high depth. There are many techniques which reduce the cache pollution caused by aggressive prefetching. The general approach is to apply filtering mechanisms on the

incoming prefetches or the outgoing prefetch requests. These mechanisms discard potentially less useful prefetches or prefetch them only to the bigger outer levels of the cache hierarchy where the negative impact of cache pollution may be low. One of the recent prefetch filters employs perceptrons to classify prefetches into useful and useless for filtering purpose [2]. The proposal uses SPP as the underlying prefetcher, turns it more aggressive, and then uses multiple weight tables indexed using different features to implement a perceptron-based prefetch filter. Depending on the sum of the table outputs compared against a predecided threshold the filter decides whether to reject or go on with a prefetch.

Prior work on prefetch filtering has employed a dynamic feedback mechanism derived using prefetcher accuracy, prefetcher timeliness, and prefetcher-caused cache pollution [10]. This feedback is used to adjust the aggressiveness of the prefetcher dynamically. This study also introduced a mechanism for estimating cache pollution and used this to decide the location in the LRU stack where a prefetched block should be inserted.

Chapter 3

ROB Stall and Load Dependents

In this chapter, we discuss how, at run time, we keep track of the ROB head stall and the number of dependents of a load that stalls the ROB head and blocks instruction retirement.

3.1 ROB Stall

ROB stall is the time for which an incomplete load instruction waits at the ROB head to complete [12]. In out-of-order-issue processors, multiple instructions run in parallel, but their results are committed in-order. The ROB is used for this purpose. It is a FIFO queue which maintains the order of instructions. An “uncommitted” result can, however, be used by any other in-flight instruction that depends on that result. ROB head is a pointer which is used to commit instructions in order. So, if there is a load instruction at the ROB head waiting for getting data from memory, the subsequent instructions cannot be committed. If the ROB is full, no more new instruction can be added to the ROB. So, the time for which an instruction waits at the ROB head to be committed is critical for performance and we will refer to it as critical delay.

3.2 Dependent Instructions

There are instructions which have a very high number of dependent instructions either in the execution pipeline or in the ROB. Such instructions are more critical for performance because they can stall the execution of many upcoming instructions. Therefore, the count of dependent instructions is an important metric for criticality along with the the ROB stall.

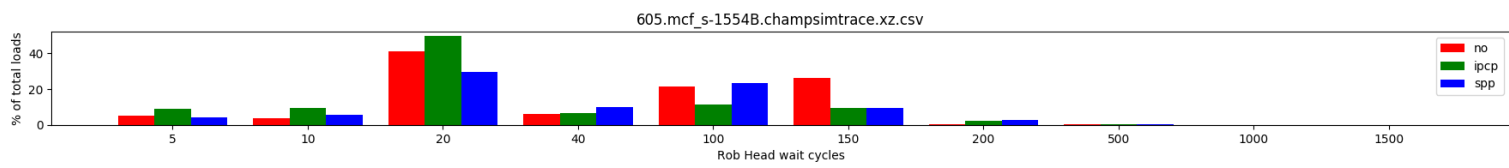
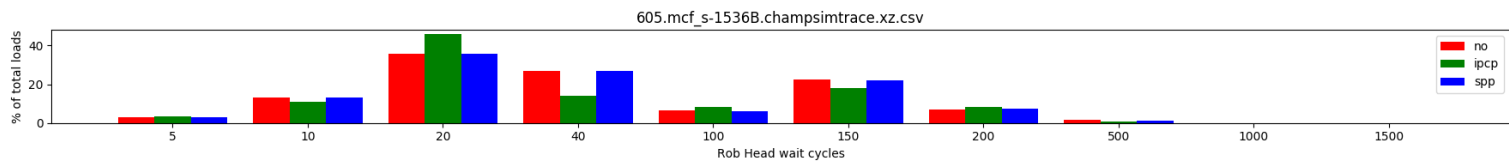
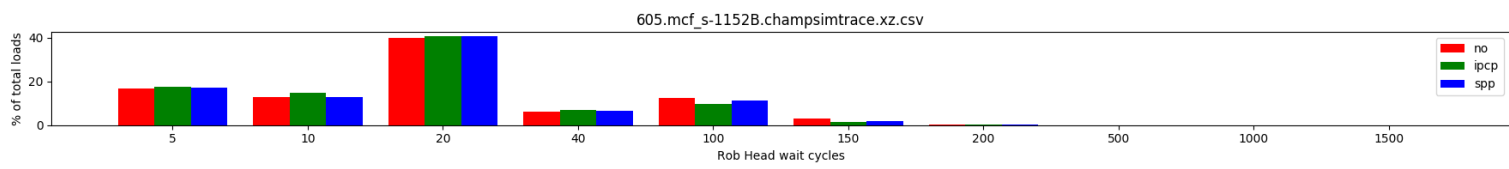
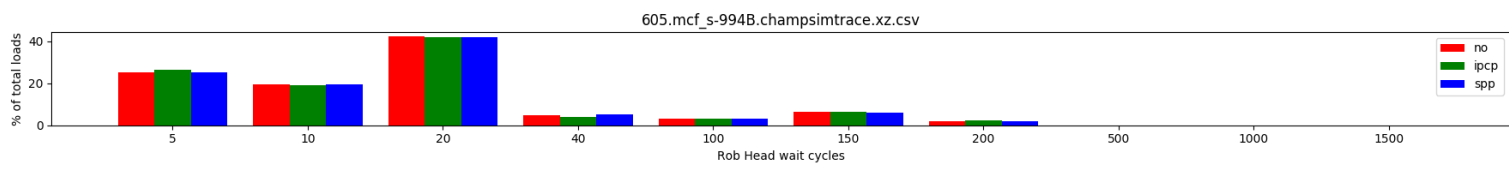
3.3 Computing ROB Delay and Dependencies at Run-time

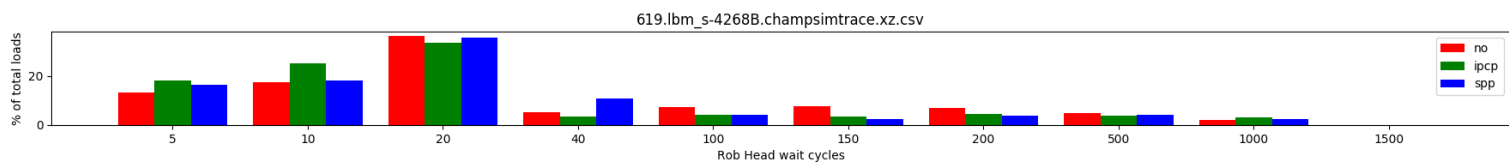
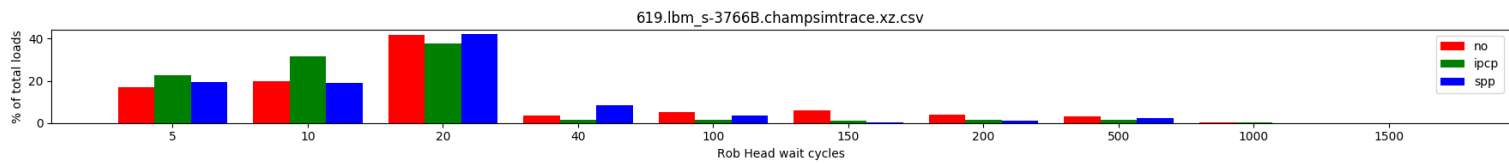
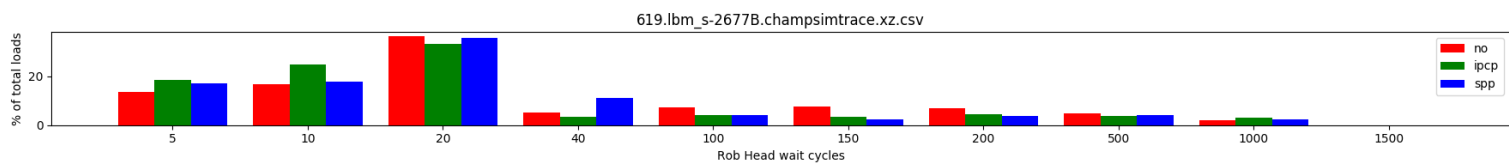
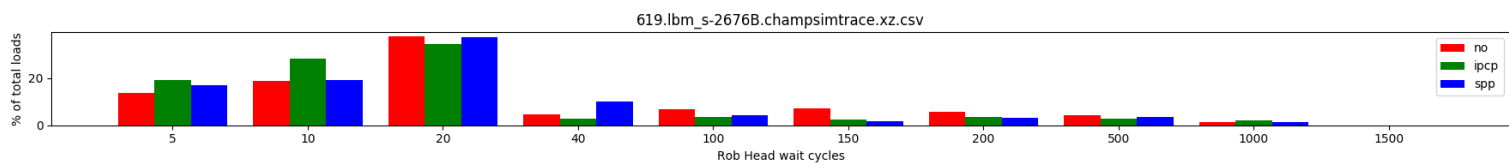
For calculating the ROB delay, we first initialize the delay to the current cycle of the simulation when an instruction is put into the ROB. Then at the time of retirement or commit from the ROB, we take the difference from the current core cycle. This is not exactly the ROB head stall, but easier to compute and a reasonable measure to identify instructions that stay in the ROB for too long. We calculate this delay for only those instructions that have at least one source operand or are load instructions. To calculate instruction dependency count for a load instruction stalled at the ROB head, we maintain the dependence tree rooted at each load instruction (the dependence tree of a load is a subtree of the bigger in-flight tree).

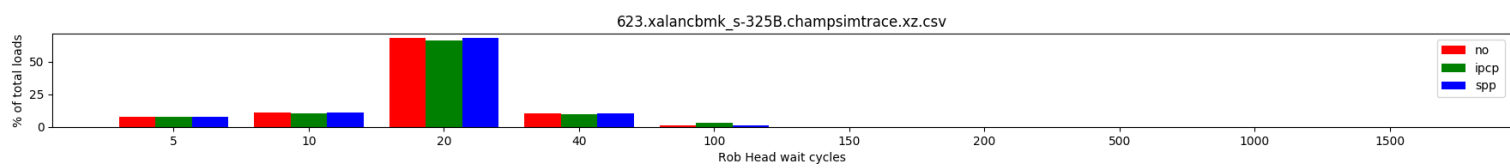
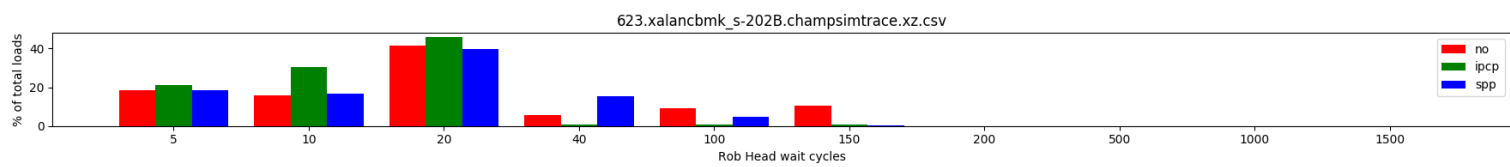
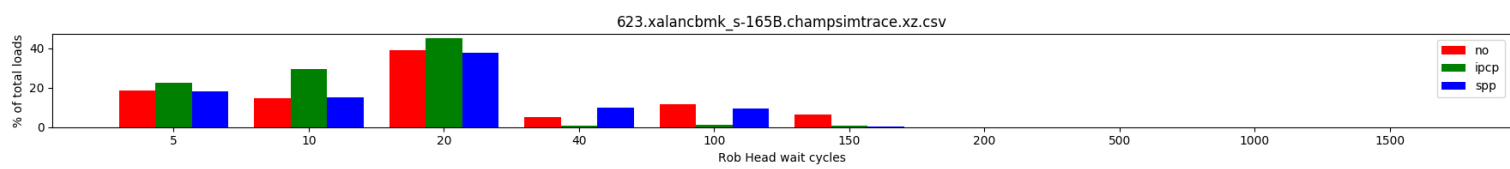
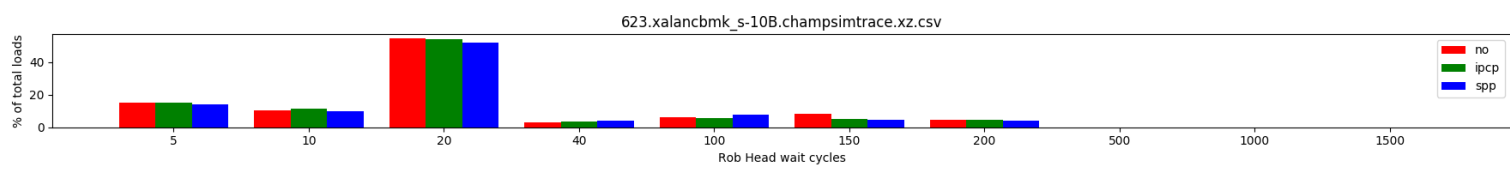
Our experimental results of ROB head stall are presented in the first two pages of Figure 3.1 (spread over the next five pages). While we collected the results for a large number of dynamic instruction traces of the SPEC CPU 2017 applications, for brevity we only show the results for selected traces of three applications, namely mcf, lbm and xalancbmk. The results are shown for three different configurations, namely without a prefetcher (no) and with two different prefetchers i.e., SPP at the L2 cache and IPCP (instruction pointer classifier-based prefetching) [13] at the L1 cache. Since 90% of load instructions have ROB head stall of one cycle, we exclude these loads from these statistics. The distribution of the remaining 10% loads is shown in these bar charts. We show the percentage of these 10% loads on the y-axis and collect statistics for a certain maximum ROB head stall. For example, the heights of the bars at the value of 20 cycles show the percentage of these loads that have ROB head stall in (10, 20] cycles. We see that most of these 10% loads have ROB head stall in the range of (10, 20] cycles. For IPCP, the percentage of loads in the smaller ROB head stall bins is higher than SPP indicating that IPCP can hide load miss latency better. This is because of two reasons. First, IPCP is an inherently better prefetcher. Second, IPCP sits alongside the L1 data cache prefetching into the L1 data cache and as a result, is better equipped to hide load miss latency.

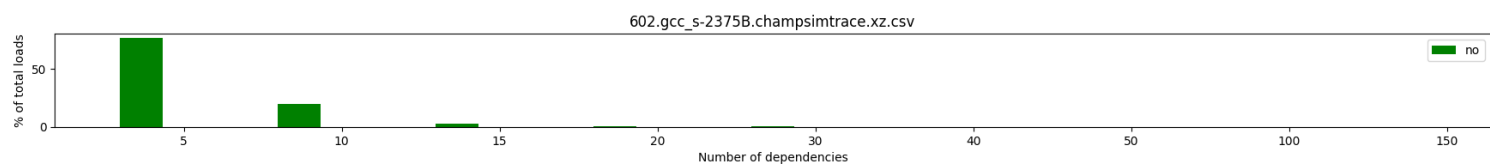
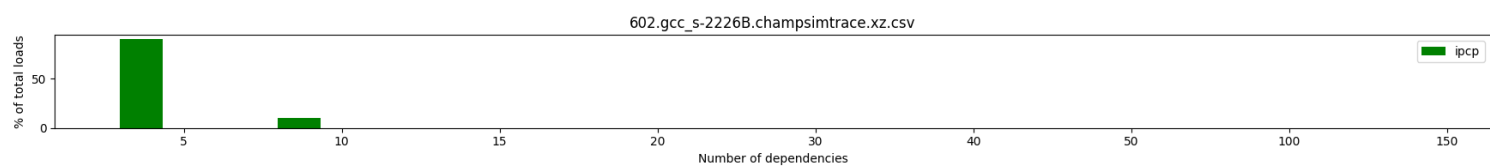
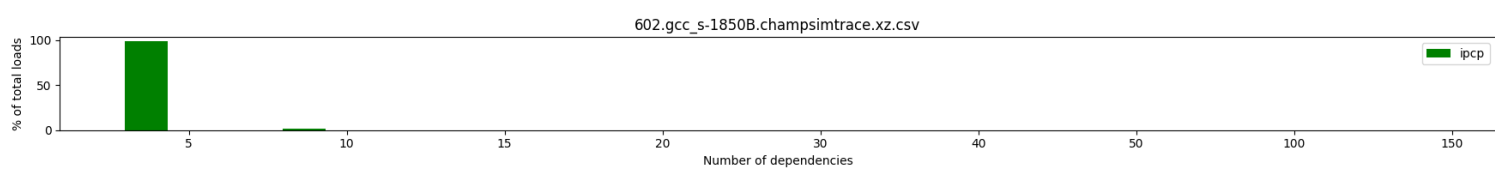
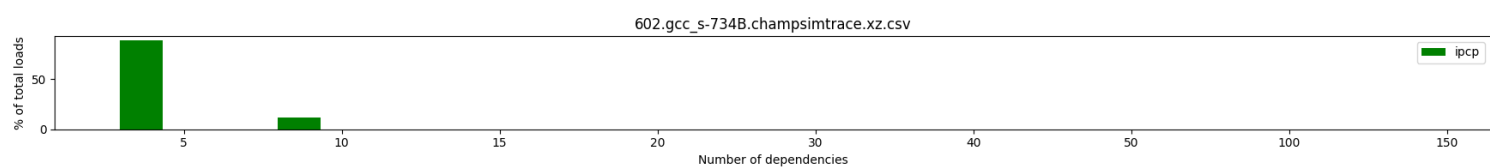
The last three pages of Figure 3.1 show the distribution of the 10% loads based on the number of dependents. We use the IPCP for collecting these statistics, although the number of in-flight dependents of a load depends only the compiled binary and the instruction window (or ROB) size. These statistics are shown for the dynamic instruction traces of three applications, namely gcc, cactuBSSN, and x264.

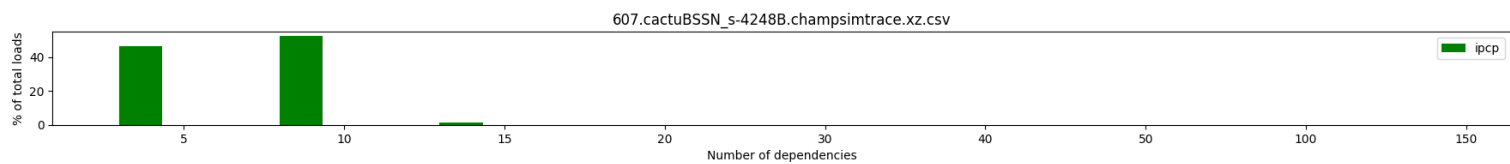
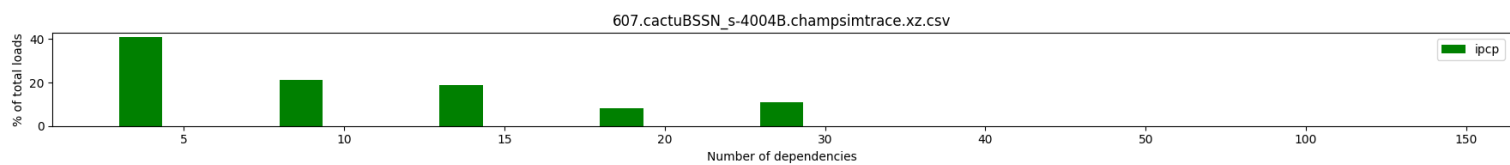
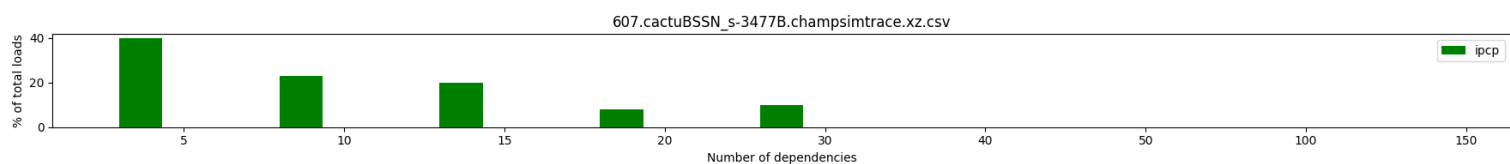
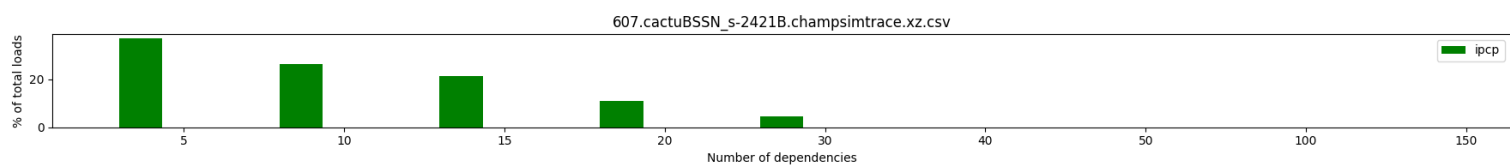
Figure 3.1: Experimental results for ROB delay and dependency (spread over the next six pages)

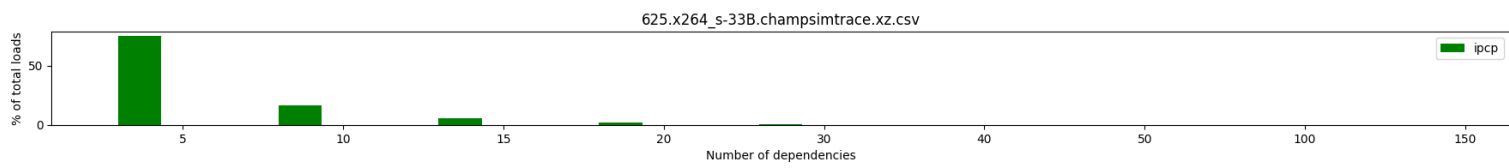
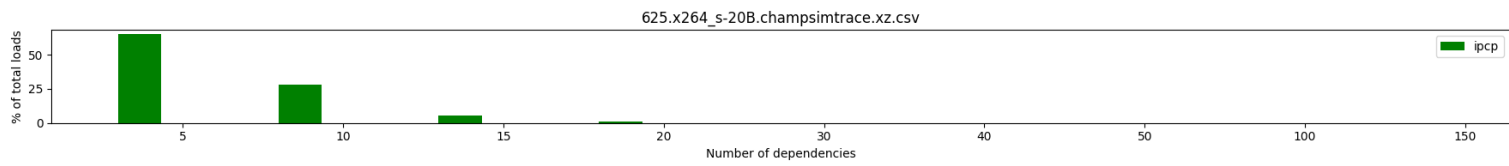
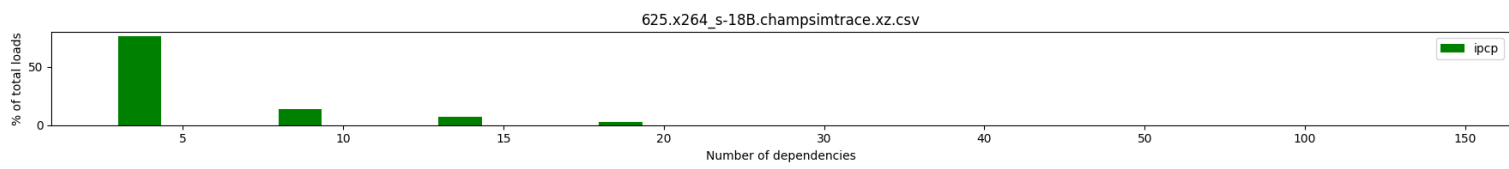
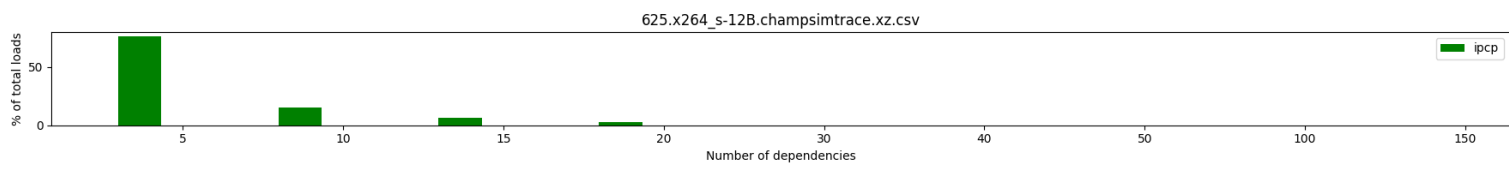












Our observations from Figure 3.1 are summarized below.

- Most of the instruction traces show ROB head stall up to 200 cycles. Some of the traces experience ROB head stall up to 500 cycles. 619.lbm_s shows ROB head stall up to 1000 cycles.
- Most of the loads from all the workloads show a dependence count of less than five.
- Most of the loads in all traces show ROB head stall in the range of (10, 20] cycles.
- There are some workloads which do not experience any improvement in ROB head stall cycles from any of the prefetchers.

3.4 Storage of Critical IPs

For keeping track of the critical load IPs and using them for prefetching, we store the IPs of the critical load instructions at run-time in a table of predefined size. The table is fully-associative and exercises LRU replacement. The table size is one of the parameters needed to be fine-tuned and we have empirically fixed it to 200 entries. Note that the physical implementation of this table can have four to eight banks each having 25-50 entries so that associative search latency of all banks in parallel can be kept within a reasonable limit. Note that we store only the IP of a critical load instruction and not the data addresses it touches. Not storing the data addresses helps reduce the size of the table. The interface of the critical IP table is shown in Figure 3.2.

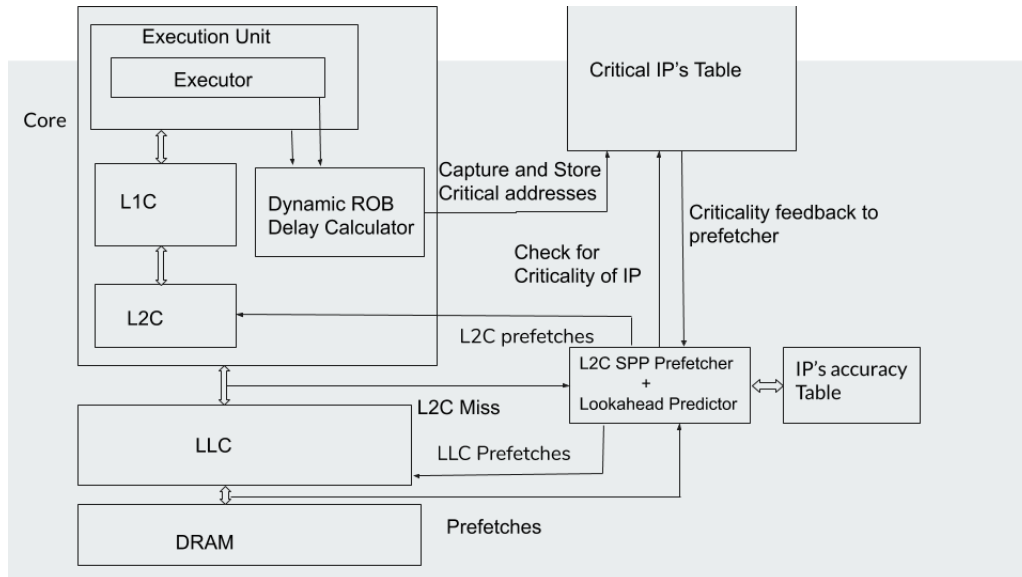


Figure 3.2: Structures to capture and use critical load IPs in a typical CPU

Once we have a table of the critical IPs, we can use this to tune prefetching. Generally, prefetchers get triggered for every miss address. So, by using this table as history of critical IPs,

we can decide for any miss address sourced by a particular load IP, whether to prefetch or not or give more importance to this IP.

There are many ways to use critical IPs for tuning prefetching decision. For example, we can trigger prefetches for every miss address sourced by only the critical IPs or use criticality information alongside SPP offering a higher lookahead depth for the critical IPs. This thesis specifically focuses on using only critical IPs for prefetching and designing lookahead mechanisms for critical loads with all other structures same as in SPP. This allows us to squarely concentrate on how to use the criticality information in prefetching without worrying about computation of delta values, or maintaining signature tables, etc.. Our primary contributions toward effective computation of lookahead depth of SPP given the set of critical IPs are discussed in the next chapter.

3.5 Determining Critical IPs

We use a combination of ROB head stall and dependence count of load instructions for deciding criticality and entry into the critical IP table. We use thresholds on ROB head stall and dependence count to decide which load instruction IPs are entered into the critical IP table. The ROB head stall varies widely across applications and across loads of a single application. This makes it very difficult to have a meaningful static threshold for ROB head stall that works for all scenarios. In fact, we found that with static thresholds, we end up having excessive prefetching for some of the applications, while some of the applications have almost no prefetching at all. Both ways, it is detrimental for performance. In general, the range of critical ROB head stall varies widely across applications and across different phases of the same application. Therefore, a mechanism is required that can dynamically adjust the critical ROB stall threshold depending on the current phase of the program and the trace.

We design an algorithm which dynamically adjusts the threshold depending upon the current behaviour of the program. We divide the range of ROB head stall in multiple bins e.g., (1, 10], (10, 20], etc. and count the frequency of each bin. We also divide program execution into intervals where the interval length is predefined in terms of the number of critical loads. At the end of an interval, we use the frequencies of the ROB head stall bins to decide the ROB head stall threshold for the next interval and reset the frequencies. Specifically, we compute the sum of the frequencies of all the bins and start accumulating the frequencies starting from the highest stall bin to the lowest. We stop when the ratio of the cumulative frequency to the total frequency is at least a predefined coverage fraction. The ROB head stall bin where we stop defines the dynamic ROB stall threshold for the next interval to decide entry into the critical IP table. The idea behind this algorithm is

to have the minimal portion of the loads as critical loads that have the ROB head stall on the higher side. We empirically find that taking the top (by ROB head stall) 1/4th to 1/7th fraction of the all loads as critical is effective. As we decrease this fraction, we start getting less number of critical addresses for prefetching and that helps decrease memory bandwidth consumption. We initialize the critical ROB head stall threshold to zero. This is found to be quite effective in having a healthy coverage to begin with, although this can put very high pressure on the cache and DRAM bandwidth in the initial learning phase.

Contrary to the variability in ROB head stall, the number of dependents of loads does not show a wide range of values. We empirically set a static threshold value of five on the dependent count.

Chapter 4

Computing Prefetch Lookahead for Critical Loads

Having decided the set of critical loads in the last chapter, we now turn to discuss the algorithm for run-time determination of prefetch lookahead of these loads. We incorporate our algorithm into SPP.

4.1 IP Accuracy-based Prefetching

Once we store all the critical loads and pass them to the prefetcher, we need to decide lookahead depths for the critical IPs. One possible method is to trace all the critical IPs and store the total number of prefetches and the corresponding number of demand hits. This gives us the total accuracy for every critical IP. Now, we can take this as a parameter to decide the prefetch depth for a given load IP.

To achieve this, we divide the IP accuracy into multiple ranges and assign a different lookahead depth to each of the ranges. Fixing this mapping empirically, however, requires significant amount of tuning effort. Even then the performance may not be good for unforeseen workloads. The biggest problem is that for certain workloads, a particular range of accuracy may not be achievable at all, thereby ruling out the lookahead depths assigned to that range. So, we need a dynamic approach which can address these issues. Additionally, we note that IP accuracy alone cannot be useful in deciding the lookahead depth because there are IPs with very high accuracy, but very few prefetches. Clearly, such IPs are not particularly useful. So, we also need to look at the volume of prefetches sourced by an IP.

One possible method is to increase or decrease prefetch depth based on the increment or decrement in IP accuracy. If IP accuracy increases by some fixed margin, we increase the depth;

otherwise it remains the same or decreases it. Another thing we did is to set a threshold for count of total prefetches. If the total number of prefetches is less than a threshold, we keep prefetching without checking the accuracy. This helps us tackle the problem of having small number of prefetches for a IP with high accuracy. It also gives effective initialization to every IP. However, deciding this threshold is also critical because if it is too high then there may be lots of prefetches even for low accuracy IPs and if it is too low then it may not be able to initialize the IPs effectively. Empirically we find out that the effective value of this threshold is close to 1000.

4.2 Using SPP Lookahead Predictor

We have also explored passing on the critical loads to SPP to see the effect of its in-built lookahead prediction mechanism. SPP uses the confidence of the predicted delta values to decide the path and lookahead depth. After every prediction, the overall confidence of the path gets multiplied by the new path confidence. When this path confidence goes below the prefetch threshold, it stops prefetching. Passing SPP only the critical loads resulted in less performance, although the L2C, LLC, and DRAM bandwidth consumption decreases relative to SPP. Also, since we are passing only critical loads, the number of addresses passed to the prefetcher is also less leading to less active energy expended in the prefetcher. Since there is no inherent mechanism in SPP to deal with criticality of the addresses, it is not able to take advantage of this information. In most of the cases, SPP is less aggressive. So, by giving more importance to critical loads and making it more aggressive can result in significant performance improvement. We have also tried mixing critical and non-critical loads in SPP. For critical loads, we predict the lookahead depth and for non-critical loads, we use SPP's lookahead depth prediction. This results in performance improvement because there are some workloads which do not have any or very low number of critical loads. This problem can also be addressed by using dynamic thresholds for ROB head stall discussed in section 3.5. These workloads are not able to take any advantage of prefetching. So, using SPP prediction for non-critical loads and our lookahead depth predictor for critical loads results in considerable performance gain with comparable bandwidth.

When using only critical IPs in SPP there are addresses which are also having threshold greater than PF_THRESHOLD. For those we can give priority to either the SPP lookahead mechanism or our own method. We experimented with both methods. By using the SPP lookahead mechanism for critical addresses having threshold greater than PF_THRESHOLD and our prediction mechanism for others results in performance improvement with less cache and DRAM bandwidth for prefetches. We call this method SPP with less bandwidth. This scheme is summarized in Figure 4.1.

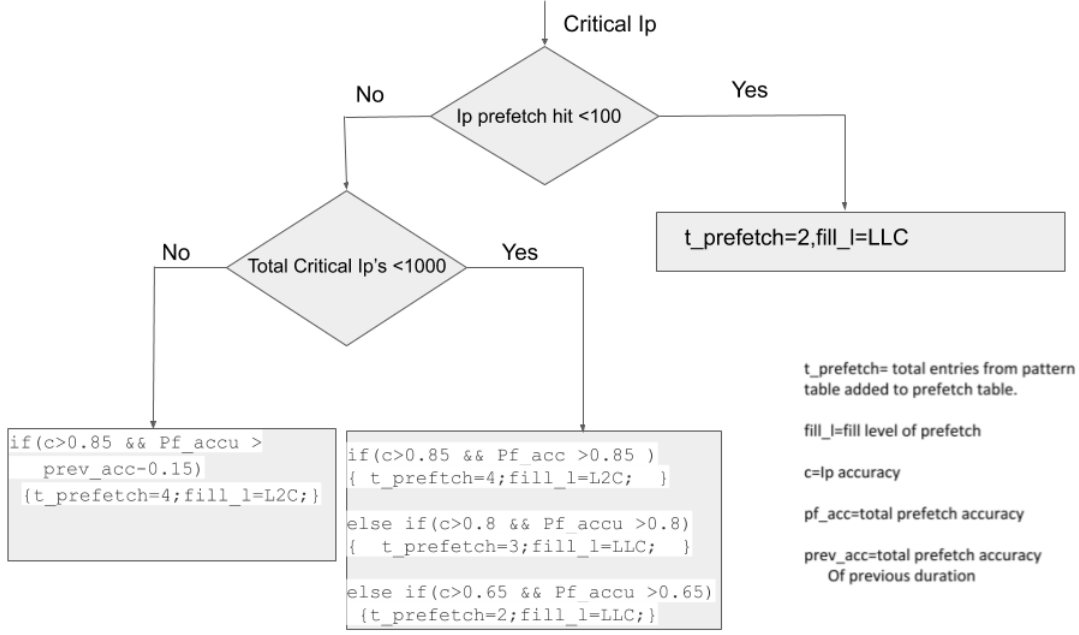


Figure 4.1: SPP with low bandwidth consumption

4.3 Prediction using Global Prefetch Accuracy

We have already discussed how local prefetch accuracy per IP can be used to decide prefetch lookahead. Another parameter we can use in deciding lookahead depth is the global accuracy of prefetching at various levels of the cache hierarchy and the overall accuracy of the prefetcher. IP-based prefetch accuracy is a local parameter and it does not take into account the overall reaction of the system to prefetching. Also, we need some parameter to check if a particular cache level is getting polluted due to aggressive prefetching. It is always desirable to have the most frequently used data in the inner levels of the cache hierarchy. However, since the inner level caches are also small in size, they can easily become polluted. At LLC we can have more prefetched data, but it is also shared. So, we also need to take that into account.

We have measured prefetch accuracy at level 2 cache (L2C), at the last level of cache (LLC) and the overall prefetch accuracy. We also keep resetting these accuracy counters at regular intervals so that they carry results of the recent history. SPP has a thresholding mechanism for every prefetch by which it decides at which level of the cache hierarchy the prefetch will be inserted. So, according to the prefetch accuracy of the cache levels, we can adjust this fill level. For example, if the accuracy of L2C goes down, we can stop prefetching to this level and insert the prefetches into LLC only. As with other parameters, deciding a threshold value for the accuracy is again a critical task. This requires multiple simulations. We find that a combination of local IP accuracy and global prefetch accuracy is desirable.

4.4 Deciding Fill Level of Prefetches

As discussed earlier, SPP prefetches into two different cache levels depending upon the threshold values of the confidence. These two thresholds are Prefetch Threshold (PF_Threshold) and Fill Threshold (FILL_Threshold). If the confidence of a prefetch is greater than the fill threshold, it is inserted into L2C and LLC; if it is greater than the Prefetch Threshold, it is inserted in the LLC only; if it is less than the prefetch threshold, it is not prefetched.

For the critical loads, we do not depend on the SPP confidence to decide the level of the prefetches. Instead, we use parameters like IP-based local accuracy and the overall prefetch accuracy for this. We start with an aggressive scheme inserting most of the prefetches into L2C and LLC. After the initial phase, all parameters start getting enough training data so that we can make predictions based on them. It is observed that most of the time L2C has high prefetch accuracy (above 90%) and LLC has around 50% to 60% prefetch accuracy. This is because most of the time, prefetched blocks are present in L2C and they receive demand hits. The prefetched blocks which are in the LLC do not get hits most of the time and get replaced by other blocks. Decision of the fill level of the prefetch is most critical to performance because prefetching a good number of accurate prefetches into the L2C significantly boosts the performance of the system.

4.5 Periodic Updates of Parameters and Initialization

As already discussed, there are multiple parameters like prefetch accuracy, ROB head stall, etc. that we need to keep track of at run-time. These parameters need to capture the behaviour of every phase of the workload to have an effective impact on performance. It is done by dividing the program or workload execution into multiple phases based on some parameters like count of critical prefetches. Once we decide this parameter, we need to decide the total length of the interval. Interval length is decided empirically by running multiple simulations with different lengths. Interval length of 1000 critical prefetches is observed to be effective.

4.6 Dynamic Thresholds

As we have pointed out, we need to manage two thresholds, namely Fill and Prefetch thresholds that decide which level an incoming prefetched block to insert into. In SPP, these thresholds are fixed to values of 90 and 25. We changed these thresholds to different values and found out that we could get better accuracy by changing these thresholds. As we know, having more accurate prefetches in the L2C can significantly improve performance. So we tried to decrease FILL_Threshold to allow more prefetches at L2C and increase PF_Threshold to discard some useless prefetches. This

resulted in increased IPC with almost the same LLC bandwidth and increased L2C bandwidth.

We also tried to dynamically adjust these thresholds based on L2C and LLC prefetch accuracy. We tried to increase or decrease these values based on the corresponding accuracy, but found out that these thresholds saturate to their extreme values and result in the same effect as static thresholds. For critical loads, we have two options. One option is to use SPP's inherent prediction mechanism and other is to use our own lookahead prediction mechanism. A possible third way is to use SPP lookahead mechanism for those workloads which have threshold at least equal to PF_THRESHOLD and our own prediction mechanism for those workloads which have threshold less than PF_THRESHOLD. This will ensure that all critical loads get lookahead depths with minimum pollution. But if we need a more aggressive prefetcher we can assign it using our own predictions. But aggressive prefetching will consume lots of cache as well as DRAM bandwidth. So, there is a tradeoff between the two.

4.7 Depth-wise and Total Prefetch-wise Approaches

There are various ways we can change the total number of prefetches based on the prediction by the prefetcher. One way is to control the total number of prefetches generated by the prefetcher and the other is to control the prefetch depth, which will indirectly control the prefetch count. In the SPP's pattern table, there are multiple entries depending upon the ways of it. We select entries from the pattern table depending upon various parameters for every depth given a critical IP. So, we basically select a subset of prefetches suggested by SPP if confidence generated by SPP is less than PF_THRESHOLD and give them new confidence; otherwise it uses confidence as in SPP and prefetches same as SPP. This results in performance almost same as SPP with less bandwidth consumption. In this low bandwidth version, to decide the threshold for fill level, we use IP-based local accuracy and overall prefetch accuracy, which every time is compared to the previous (interval)-(offset) to get the dynamic estimates. We also prefetch into the LLC alone in the case when the number of critical IPs seen so far is less than 1000 to initialize and train all the parameters properly. Controlling the depth results in aggressive prefetching because at every depth we prefetch two or more blocks depending upon the number of ways in the SPP pattern table. We go beyond the depths used by SPP. In this version, we directly put the prefetches in the L2C to get the more aggressive prefetching and use a threshold on IP-based and L2C accuracy to decide depth of prefetching. Controlling the number of prefetches is difficult but results in less bandwidth consumption and less aggressive prefetching compared to controlling depth. Our depth prediction mechanism integrated with SPP is shown in Figure 4.2.

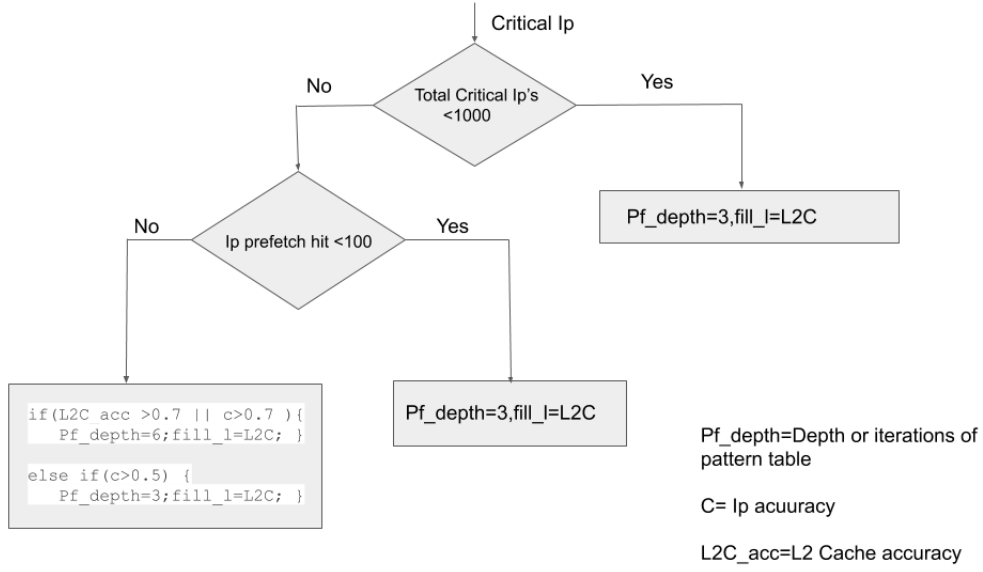


Figure 4.2: SPP with depth prediction configuration

We have done experiments with different depths and with prediction of the number of prefetches. By Varying depths, we observe that increasing depth increases bandwidth consumption and IPC up to a certain point after which it starts decreasing due to bandwidth constraints. This analysis help us in various ways. First, it shows the effect of prefetching at various degrees (from less aggressive to more aggressive). Along with that, we are able to learn the effect of pollution at various cache levels and their effect on performance.

Chapter 5

Experiments and Results

We have experimented with a large number of configurations using the ChampSim simulator. In the following, we will discuss the details of the results. The highlights include performance improvement on top of SPP in single-core high bandwidth configuration and reduction in prefetch bandwidth at the DRAM and the LLC while maintaining SPP performance. In multiprogrammed workloads, our proposal delivers performance close to SPP while consuming less bandwidth. We will also discuss experiments with varying prefetch depths and threshold values offering us more insights into the behaviour of SPP and the proposed improvements.

The following prefetcher configurations are evaluated. These prefetcher configurations are used only at L2C and there is no prefetcher at L1C and LLC. These are evaluated for single-core as well as quad-core environments. For the quad-core environments, we will report results for homogeneous (rate mode) and heterogeneous multiprogrammed workloads.

- Baseline SPP
- SPP working with only critical load addresses
- SPP with changed threshold
- Enhanced SPP lookahead mechanism with less bandwidth proposal
- Enhanced SPP lookahead mechanism with higher bandwidth proposal (depth version)

To evaluate the results, we use the following metrics.

- Geometric mean of IPCs over all the traces
- Average L2C, LLC, and DRAM Bandwidths over all the traces
- Average number of addresses input to the prefetcher over all the traces
- Average accuracy of the prefetches at the L2C and LLC over all the traces.

- Comparison of the depth-critical Workloads individually

Depth-critical workloads are those which experience significant increase in IPC when depth or total number of prefetches increases. In the following results, cache and DRAM bandwidth means the total number of prefetch requests to the cache and DRAM, respectively. For example, DRAM bandwidth means total number of prefetch requests from LLC to DRAM.

For high-bandwidth configurations, we set the pattern table way (PT_WAY) to 2 instead of 4 because in this case, the prefetcher explores higher depths and we get fine-grain control over the total number of prefetches generated by having a lower value of PT_WAY. In the simulations of multi-programmed workloads, we set the threshold for dynamic ROB head stall to one-seventh of the largest delay from the last interval. For single-core simulations, this is set to one-quarter of the highest delay. This is done because of increased bandwidth consumption and cache pollution in multi-core configurations.

5.1 Single Core Results

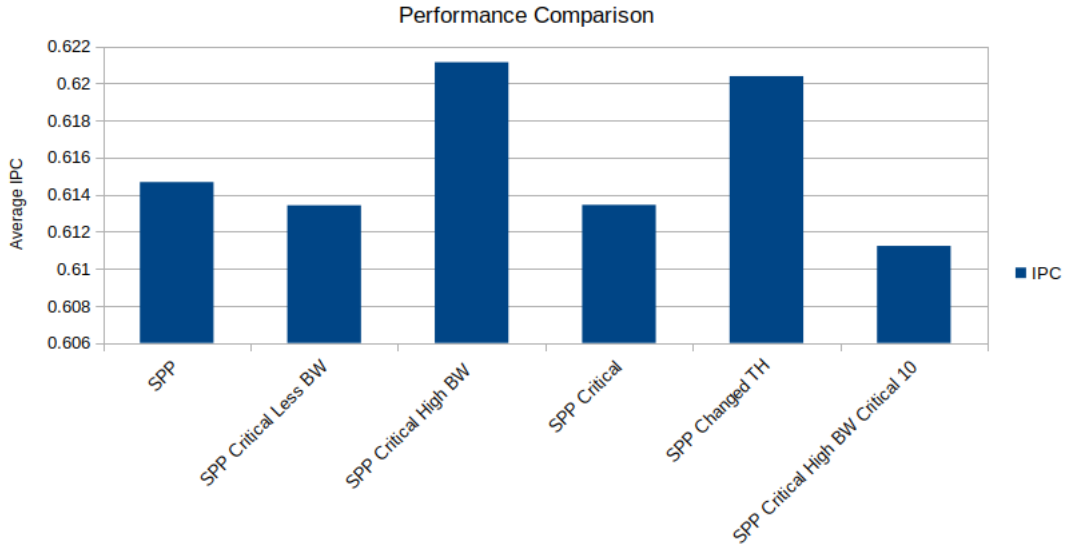


Figure 5.1: Performance comparison for single-core configuration

Figure 5.1 shows the single-core performance for different configurations, while Figures 5.2, 5.3, and 5.4 respectively show the number of prefetch lookups at the L2C, LLC, and DRAM. For single core, our less bandwidth proposal (SPP Critical Less BW) performs similar to SPP with 4-5% less LLC and DRAM bandwidth consumption. If we pass only critical load addresses to the SPP (SPP Critical), we get approximately same performance but with less L2C and LLC bandwidth consumption.

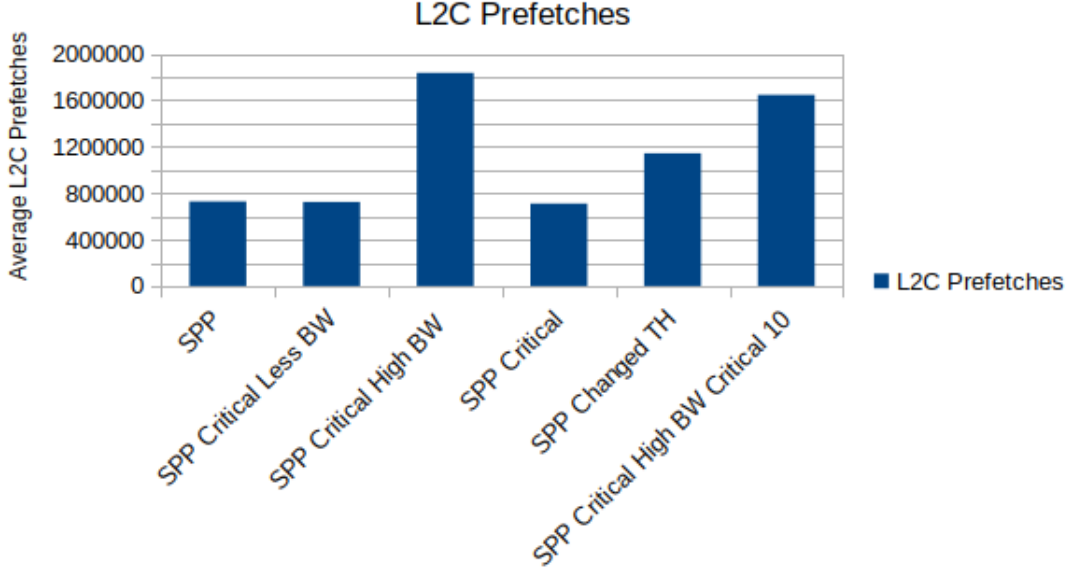


Figure 5.2: Number of prefetch lookups at L2C for single-core configuration

In the high bandwidth proposal (SPP Critical High BW), we prefetch using higher depths for every source address. Compared to SPP, we get about 1% improvement in IPC, but require almost $2.5\times$ more L2C bandwidth and 36% more LLC bandwidth. If we compare it with a configuration where for every address passed to SPP, it prefetches till depth ten (SPP Critical High BW Critical 10) then we get only 0.2% performance improvement while consuming 11% more L2C bandwidth but with 41% and 40% less LLC and DRAM bandwidth respectively. In some cases, our high bandwidth prefetching proposal goes till depth 12 and because of this, we consume more L2C bandwidth in these cases. However, overall, our high bandwidth proposal is able to convert this bandwidth consumption into reasonable performance improvement (average 1% IPC improvement is significant considering the large number of traces).

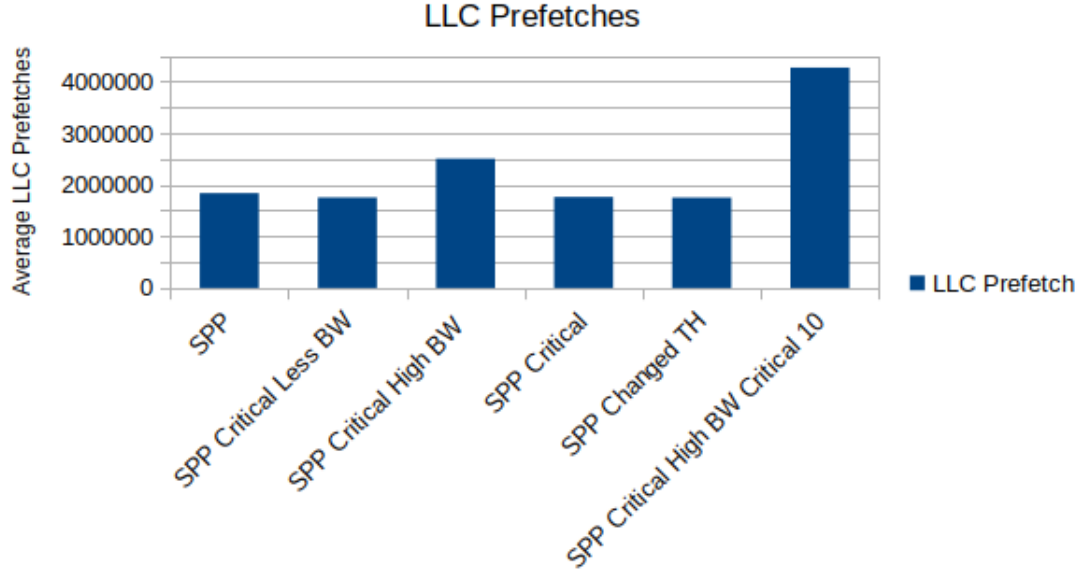


Figure 5.3: Number of prefetch lookups at LLC for single-core configuration

We also experimented by varying the PF_Threshold and Fill_Threshold in SPP. These results are shown in the bar marked “SPP Changed TH”. If we increase PF_Threshold by 5 and decrease Fill_Threshold by 5 in SPP, it results in almost 1% performance improvement while requiring 57% more L2C prefetch lookups and about 5% less LLC and 6% less DRAM prefetch lookups. This allows more prefetches at L2C, but prefetched blocks with lower confidence are filled in the LLC only. However, it still require all the load addresses to be passed to the prefetcher as in SPP, thereby increasing the prefetcher activity and energy expense quite significantly.

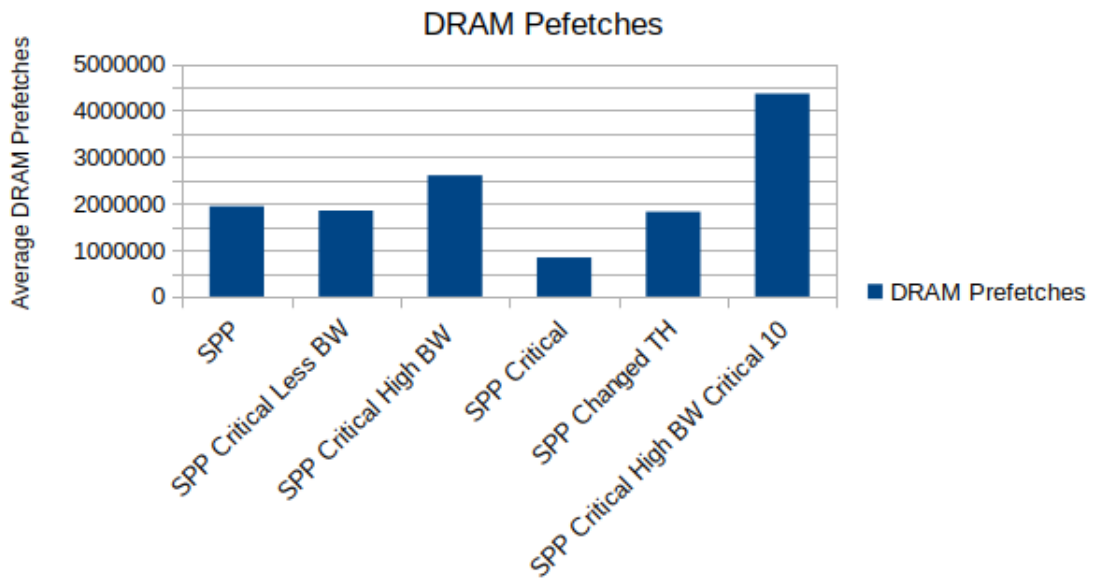


Figure 5.4: DRAM prefetch request count for single core configuration

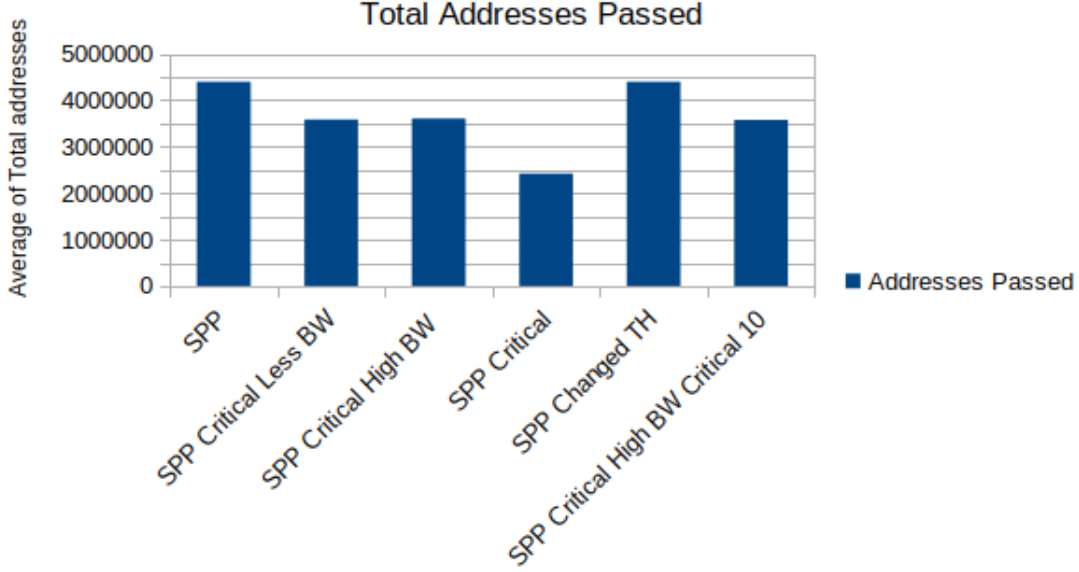


Figure 5.5: Number of load addresses passed to SPP for single core configuration

To understand the savings in the activity of the prefetcher due to criticality-based filtering, Figure 5.5 shows the number of load addresses passed to the prefetcher for various configurations. Because SPP uses more sophisticated lookahead mechanism, when we pass only the critical load addresses to it (SPP Critical), it results in 56% less DRAM Bandwidth consumption and 45% less total addresses passed to the prefetcher with slightly lower IPC compared to the baseline SPP. The baseline SPP processes 22% more input addresses compared to the less bandwidth and high bandwidth proposals.

Figure 5.6 shows the accuracy of the prefetches as seen by the L2C and LLC. The prefetch accuracy at L2C is close to 96% for the less bandwidth proposal and 85% for the high bandwidth proposal. This is an expected result because the less bandwidth proposal is more selective about prefetching.

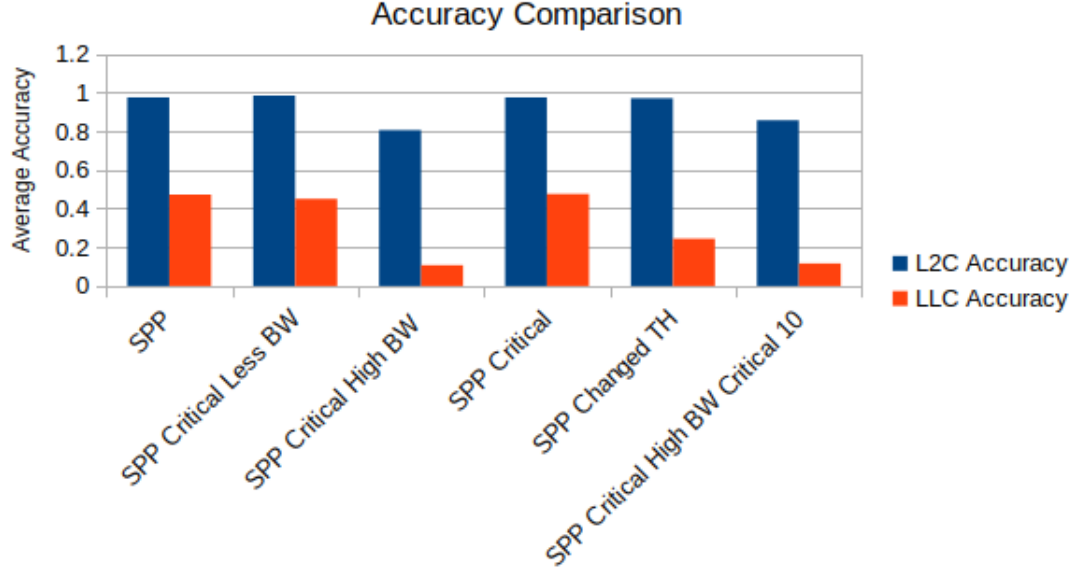


Figure 5.6: Accuracy of prefetches at L2C and LLC for single core configuration

The prefetch accuracy at LLC is around 45% for the less bandwidth proposal and around 10% for the high bandwidth proposal. The LLC accuracy is low because most of the prefetched blocks get inserted in both L2C and LLC and most of the demand hits are served by L2C obviating the need to look up the LLC. Additionally, the high bandwidth proposal is likely to pollute the caches with a large number of inaccurate prefetches.

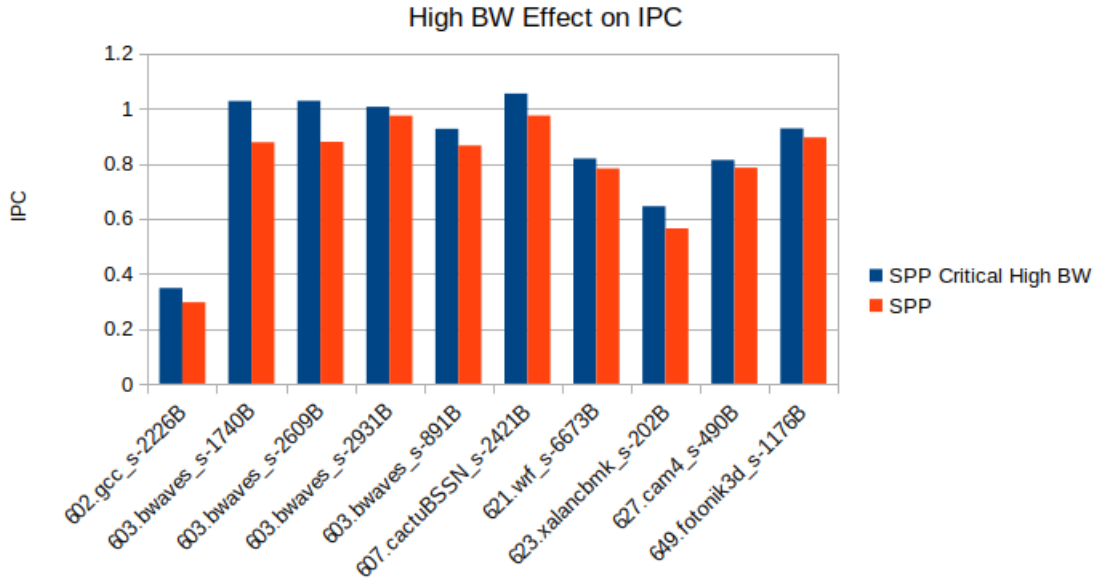


Figure 5.7: Analysis of high bandwidth proposal's effect on IPC for single core configuration

Since our high bandwidth critical load proposal improved the average performance quite reasonably, we examine the individual workloads which enjoy significant performance improvement

with this proposal in Figure 5.7. These workloads continue to inject more useful prefetches at higher depths. Performance can be further increased if we use some kind of filtering of prefetches that can reduce the pollution, keep the good prefetches, and increase the prefetch accuracy.

5.2 High Bandwidth Proposal with Different Depths

To see the behaviour of SPP with critical addresses at high bandwidth and to find out the optimal depth, we conducted experiments with different depth values. In these experiments, we predict the fill level of the prefetches and no address is dropped; all of them insert prefetched blocks at L2C or LLC depending on the fill level.

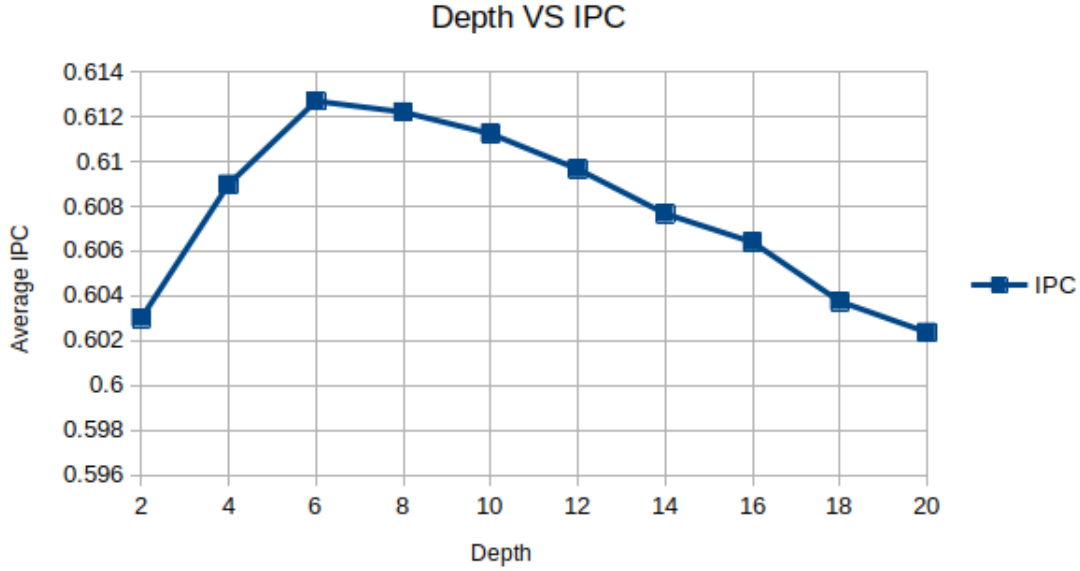


Figure 5.8: Depth vs. IPC for single core configuration

Figure 5.8 shows the variation in IPC with depth. We can see a skewed bell curve with its peak at depth six. We see steep increases in IPC from 2 to 4 depth and 4 to 6 depth. This may be due to the fact that there are many workloads which take advantage of increased prefetch depths. After that the average IPC starts decreasing, although certain individual workloads continue to enjoy improved performance which is why the decline in average IPC is rather slow.

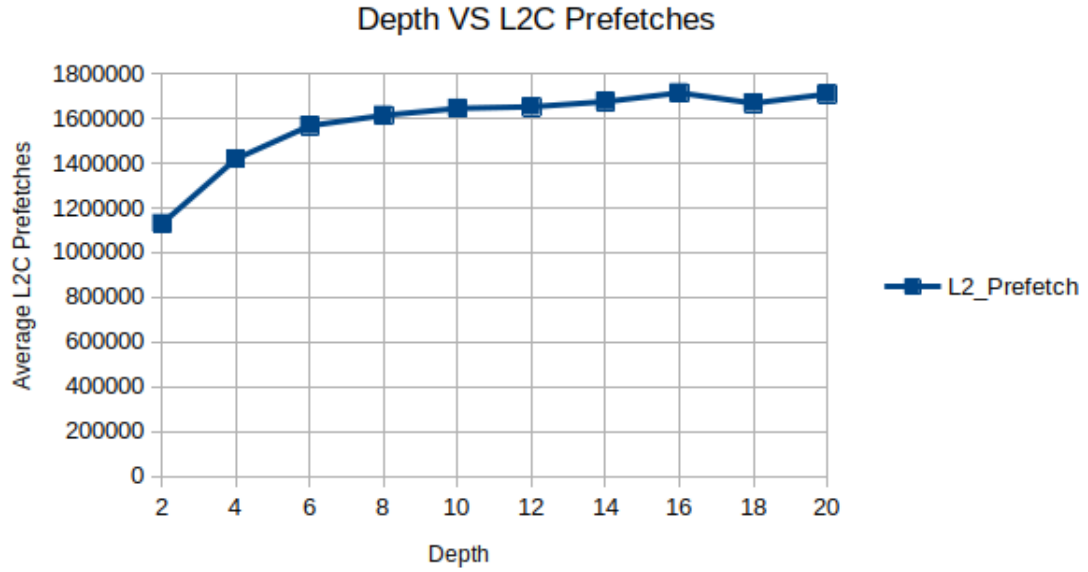


Figure 5.9: Depth vs. L2C prefetch count for single core configuration

Figure 5.9 shows the number of L2C prefetched block insertions as a function of depth. L2C prefetch count increases as we increase depth and after a point it becomes nearly a constant. It is because the parameters we use like accuracy of prefetches at any level start decreasing and once they go below a threshold value, the prefetches are blocked from inserting into the L2C.

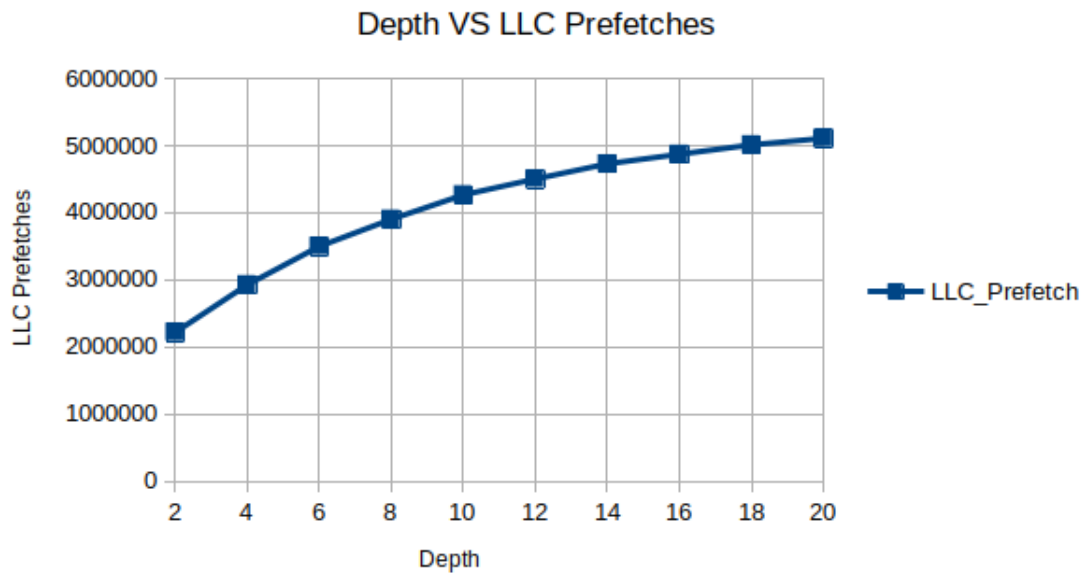


Figure 5.10: Depth vs. LLC prefetch count for single core configuration

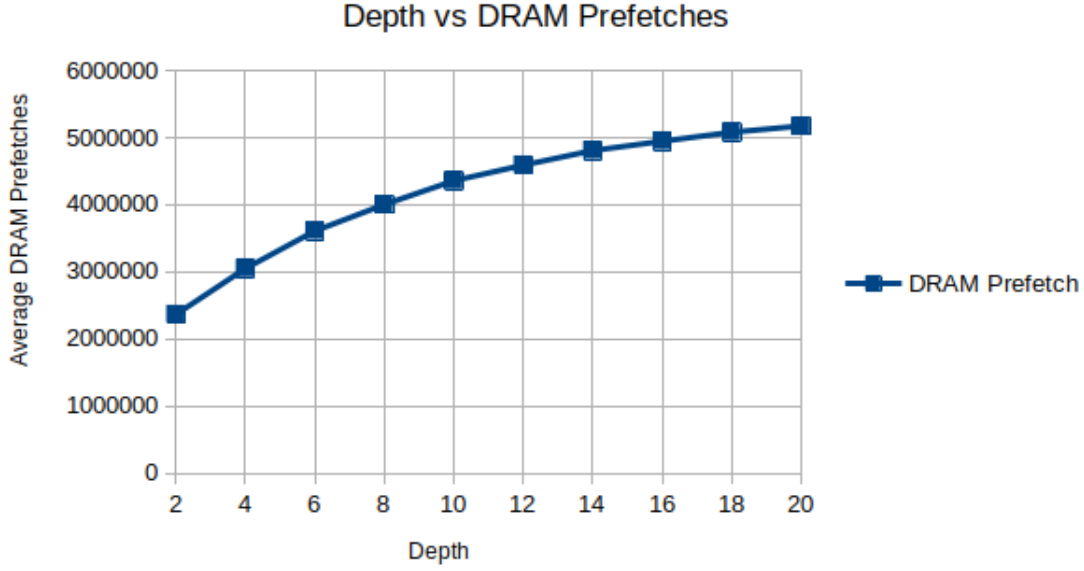


Figure 5.11: Depth vs. DRAM prefetch count for single core configuration

Figures 5.10 and 5.11 respectively show the LLC and DRAM prefetch counts as a function of prefetch depth. These functions have the shape of a quadratic. Because parameters at LLC have lower threshold values than at L2C, a much higher depth is needed for the prefetch count to start declining. These thresholds decide whether to insert a prefetched block at a cache level or not.

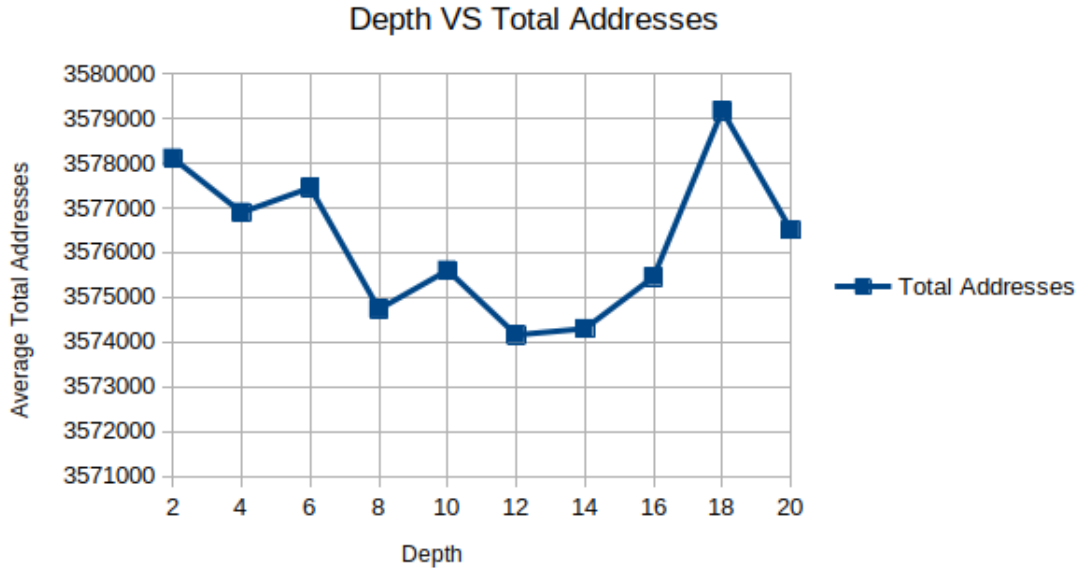


Figure 5.12: Depth vs. number of addresses input to the prefetcher for single core configuration

Figure 5.12 shows the number of addresses input to the prefetcher as a function of depth. We can see some variation in total number of addresses passed to the prefetcher (only critical addresses are passed). The variation arises primarily due to cache pollutions, but there are other factors also.

The more misses a workload generates, it will make other IPs wait at the ROB head leading to generation of more critical loads and more prefetches as the depth increases.

5.3 Quad-core Homogeneous Results

Each core has private L2C. The LLC is shared among the cores. The shared LLC can become congested for higher bandwidth demands. In the following, we discuss the results for homogeneous multi-programmed workloads where all four cores run the same Workload.

Figure 5.13 shows the average IPC of different configurations, while Figures 5.14, 5.15, and 5.16 quantify the number of prefetches respectively at L2C, LLC, and DRAM. We can see that our high bandwidth proposal generates lots of L2C prefetches ($2.5\times$ more than SPP), LLC prefetches (36% more than SPP), and DRAM prefetches (31% more than SPP). Therefore, the homogeneous multi-core workloads were not able to take advantage of aggressive prefetching. The performance of the high bandwidth proposal is 2.6% lower than SPP.

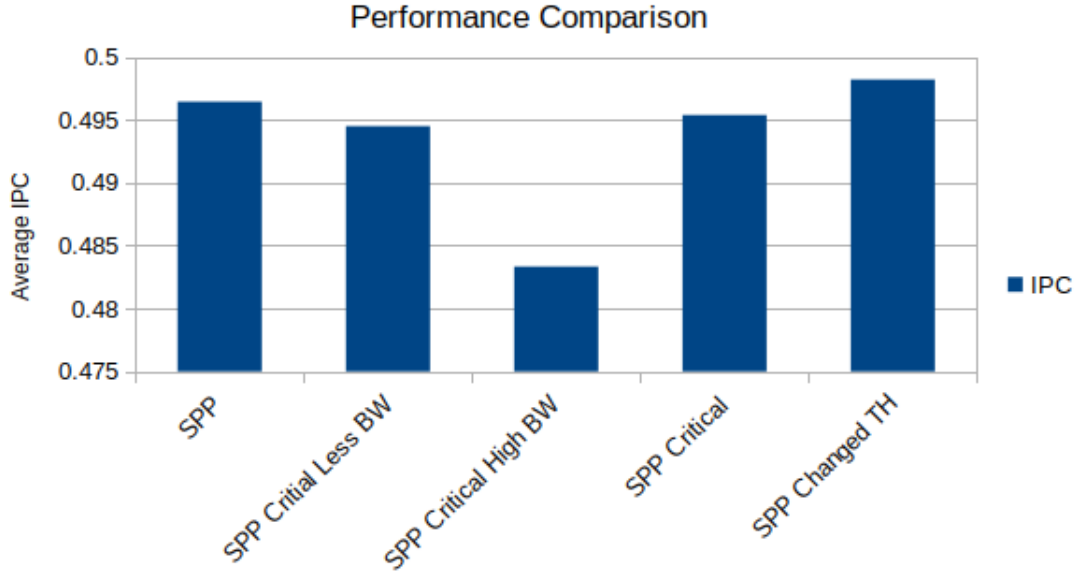


Figure 5.13: Performance comparison for quad-core homogeneous workloads

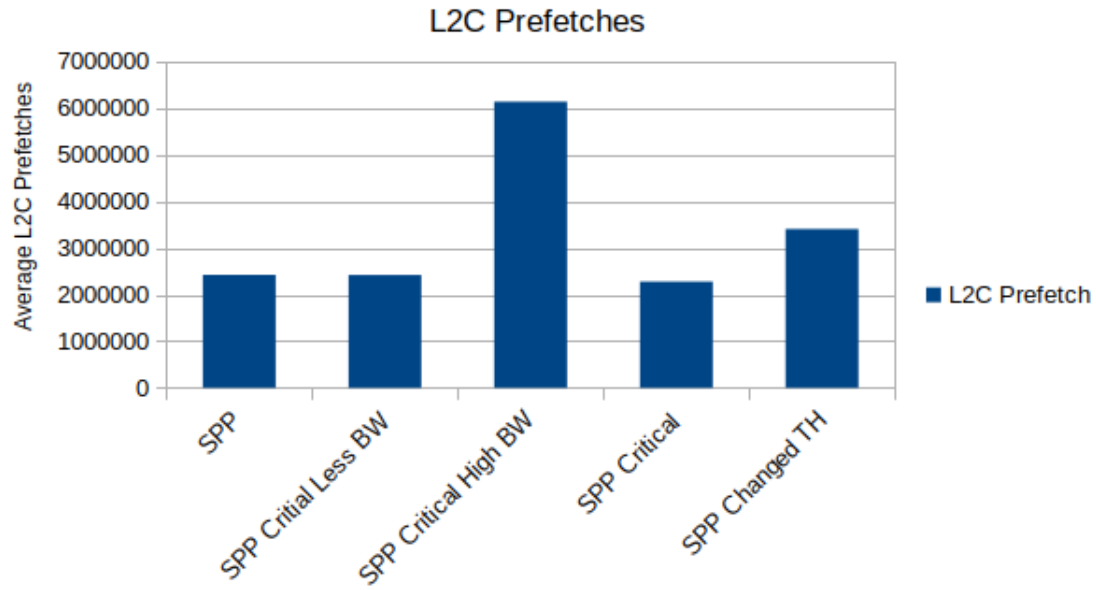


Figure 5.14: Number of L2C prefetches for quad-core homogeneous workloads

The low bandwidth proposal results in approximately the same L2C prefetches and approximately 5% less LLC and 5.6% less DRAM prefetches. Also, its performance is only 0.3% lower than SPP.

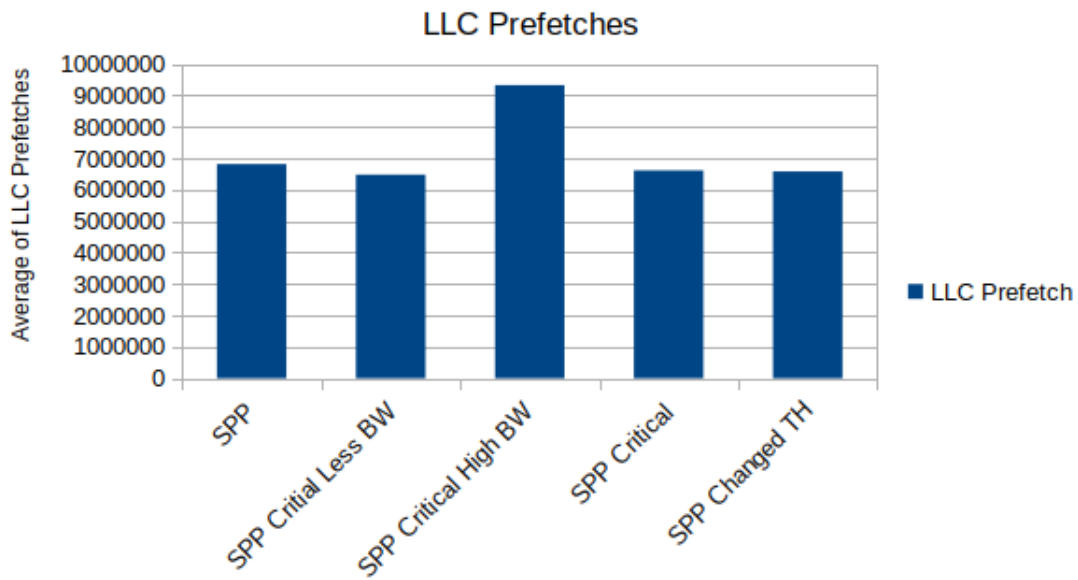


Figure 5.15: Number of LLC prefetches for quad-core homogeneous workloads

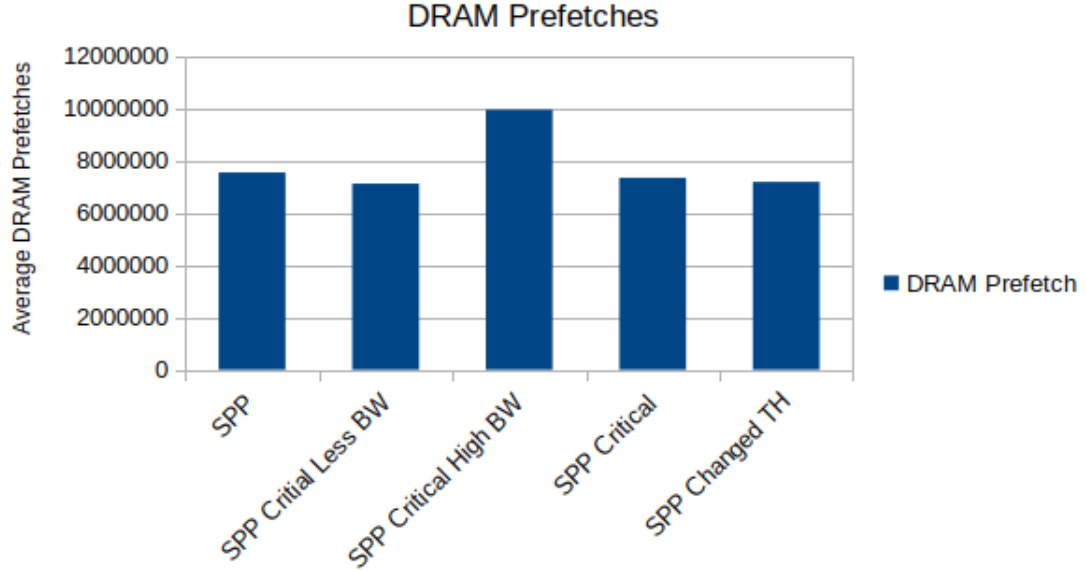


Figure 5.16: Number of DRAM prefetches for quad-core homogeneous workloads

SPP with only critical addresses behaves almost similarly as SPP with 0.2% lower performance and 6% less L2C and 3% less LLC and DRAM prefetches.

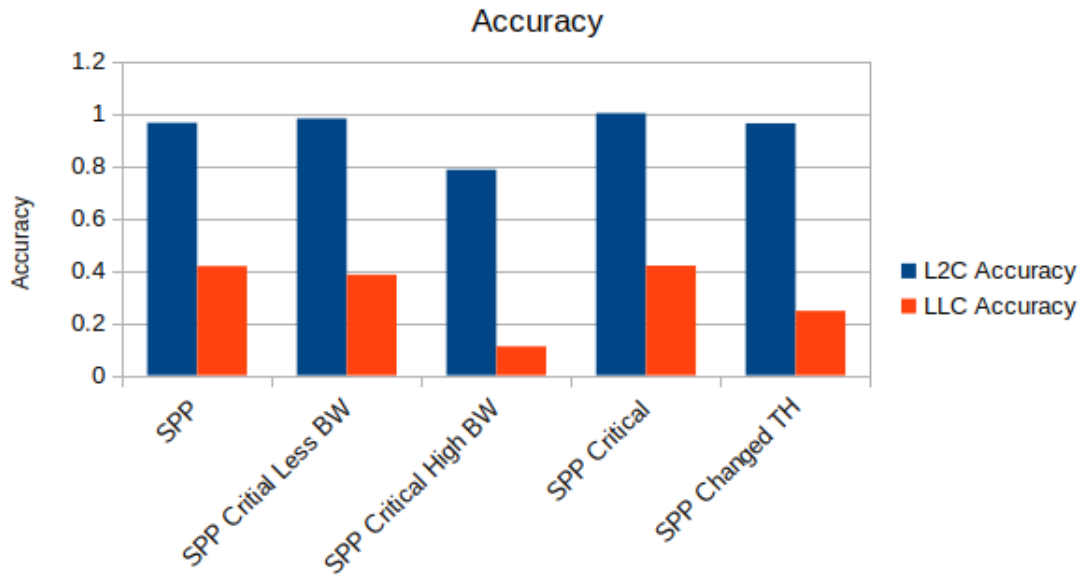


Figure 5.17: Accuracy comparison for quad-core homogeneous workloads

Changing PF_THRESHOLD and FILL_THRESHOLD gives us 0.3% performance improvement over SPP with 40% more L2C prefetches, 3.4% less LLC and 5% less DRAM prefetches. This is because even though we allow more prefetches in L2C and less in LLC, the internal SPP lookahead mechanism throttles the prefetch lookahead more efficiently. It is also because SPP has more addresses and can learn more efficiently. The number of addresses passed to the prefetcher

is 20% less in SPP with critical addresses than in SPP (Figure 5.18). Also, due to high pollution in the high bandwidth proposal, it has lower L2C and LLC prefetch accuracy (Figure 5.17).

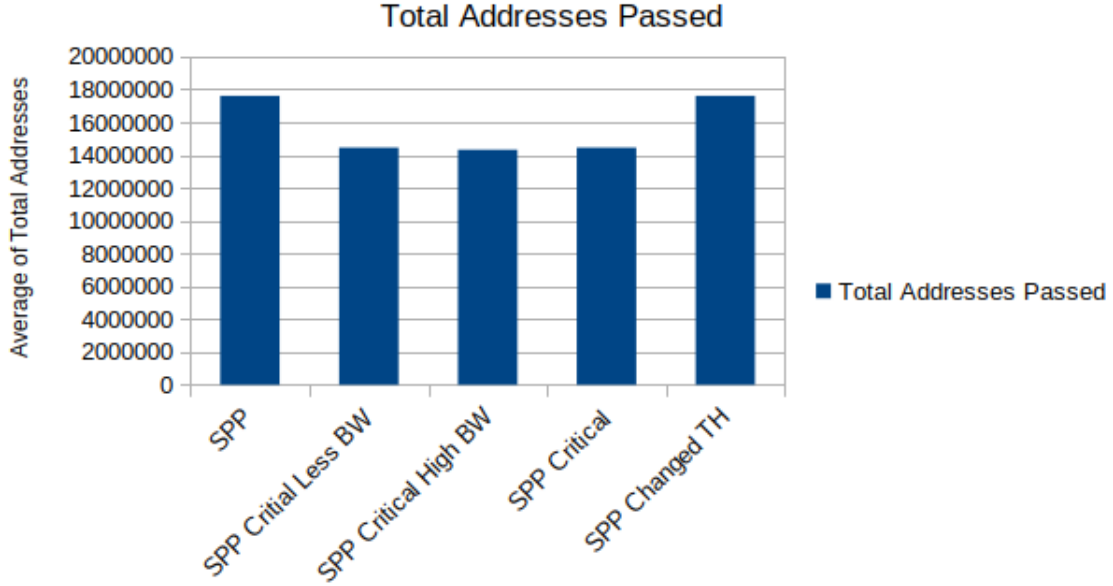


Figure 5.18: Number of addresses input to the prefetcher for quad-core homogeneous workloads

5.4 Quad-Core Heterogeneous Workloads

We combine four random single-core workloads to build 95 different heterogeneous quad-core workloads. Every single-core workload occurs four times in the combinations ensuring unbiased representation of all the single-core workloads.

Figure 5.19 shows the performance, while Figures 5.20, 5.21, and 5.22 quantify the number of prefetches respectively at L2C, LLC, and DRAM. Performance for the high bandwidth proposal is approximately 1.6% lower than SPP. It consumes almost $3\times$ more L2C bandwidth than SPP. However, it consumes only 10% more LLC and DRAM bandwidth than SPP. Unlike in homogeneous workloads, in heterogeneous case the cores compete for LLC for different data in all configurations. The less bandwidth proposal consumes 1.4% less L2C bandwidth, 10% less LLC bandwidth and DRAM prefetches, and results in 1.1% lower performance than SPP. We have also experimented by changing the pattern table ways from 4 to 2 for other configurations apart from the high bandwidth and it resulted in less performance reduction. But we have presented the result for PT_WAY of 2 for having uniform parameter settings across all results. By changing the threshold values we get 0.5% more performance with 46% more L2C bandwidth, 1.8% less LLC and 2.4% less DRAM prefetches than SPP.

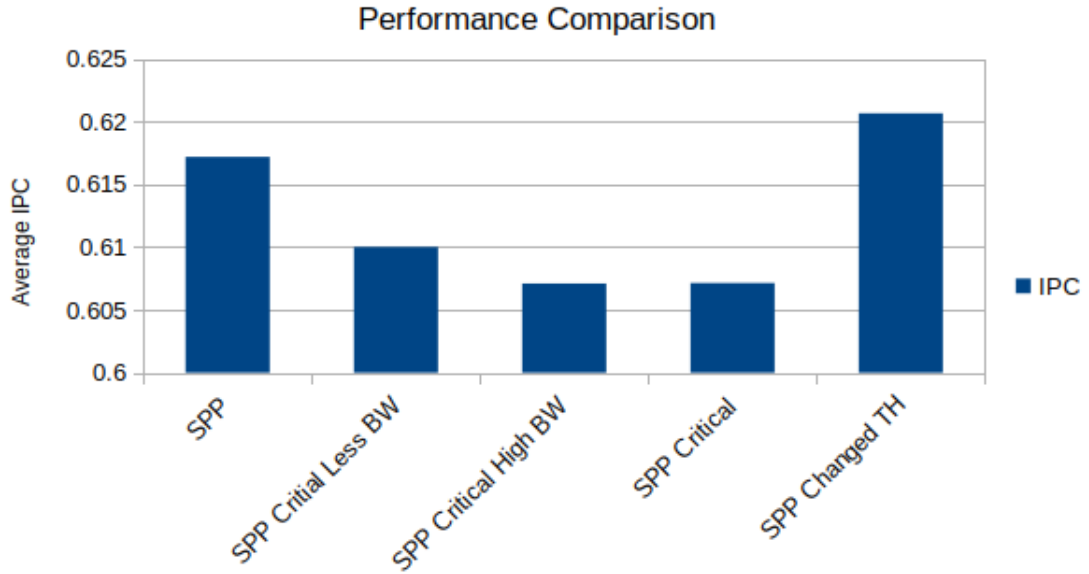


Figure 5.19: Performance comparison for quad-core heterogeneous workloads

The LLC and DRAM prefetch counts for SPP are very high for heterogeneous workloads, Whereas the less bandwidth proposal and SPP Critical configuration have approximately 10% less LLC and DRAM prefetches.

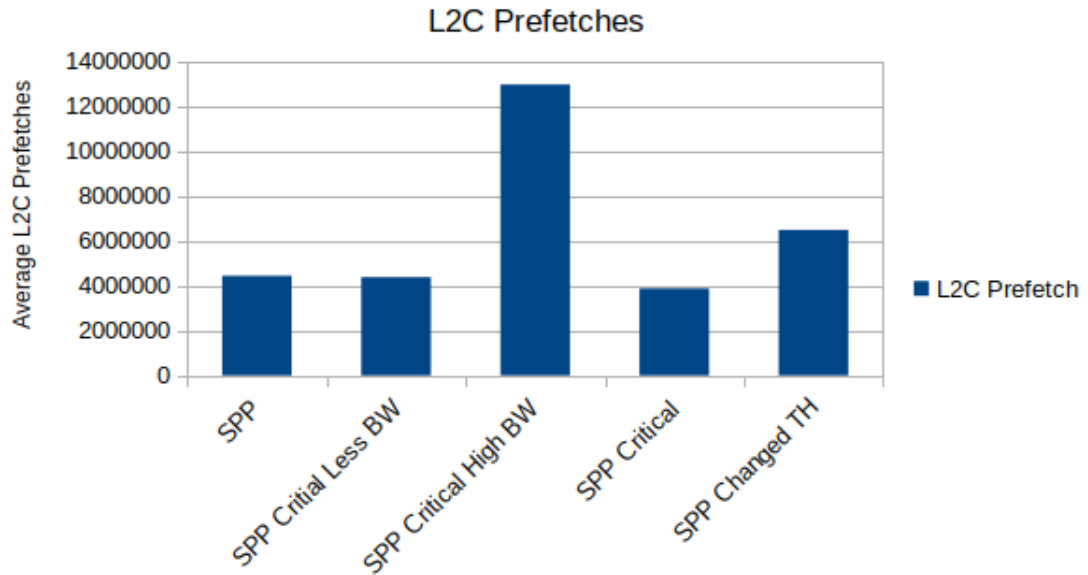


Figure 5.20: L2C prefetch count comparison for quad-core heterogeneous workloads

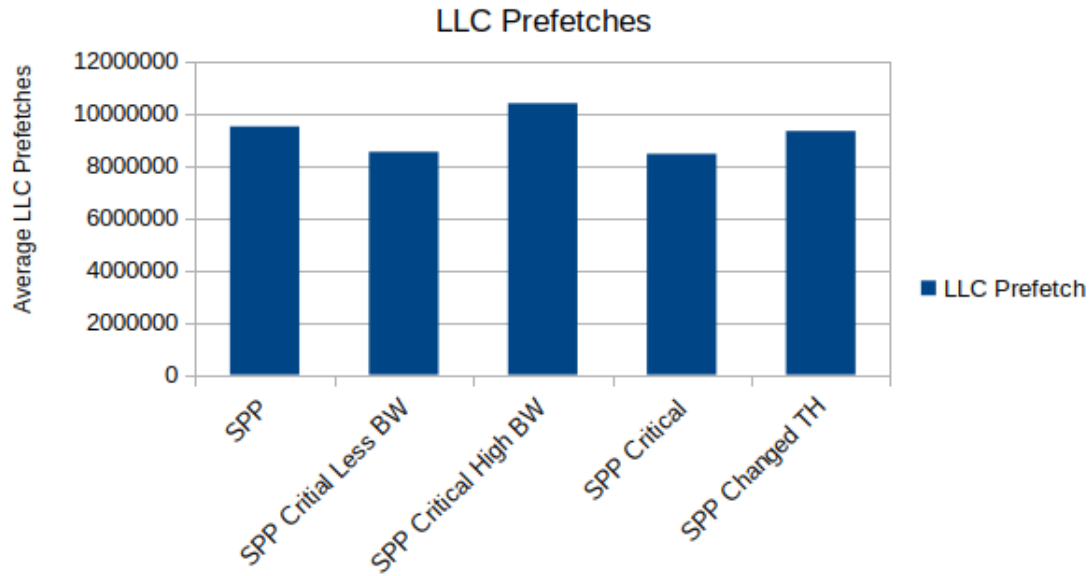


Figure 5.21: LLC prefetch count comparison for quad-core heterogeneous workloads

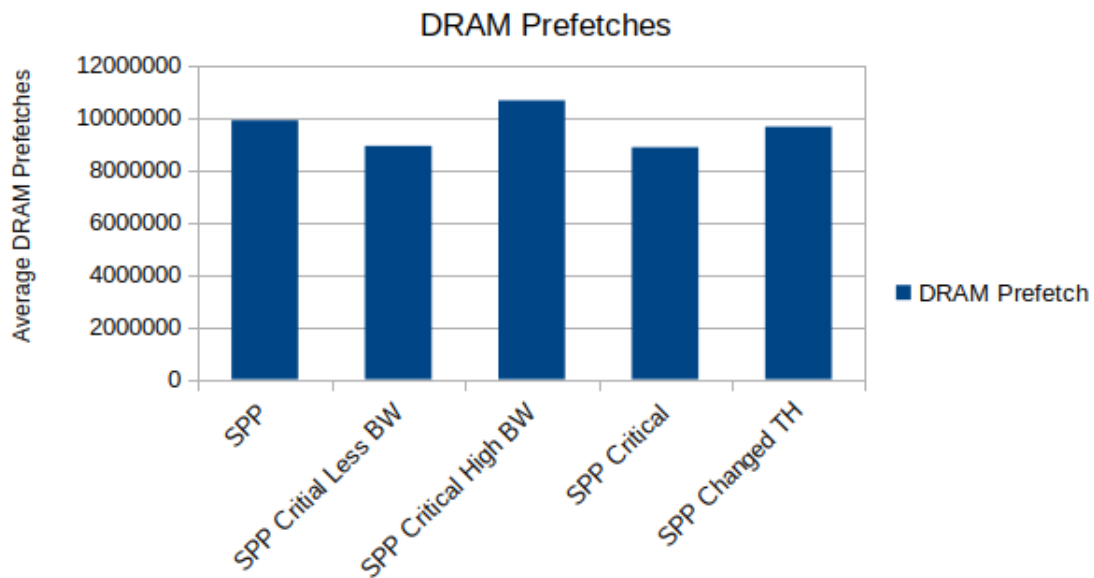


Figure 5.22: DRAM prefetch count comparison for quad-core heterogeneous workloads

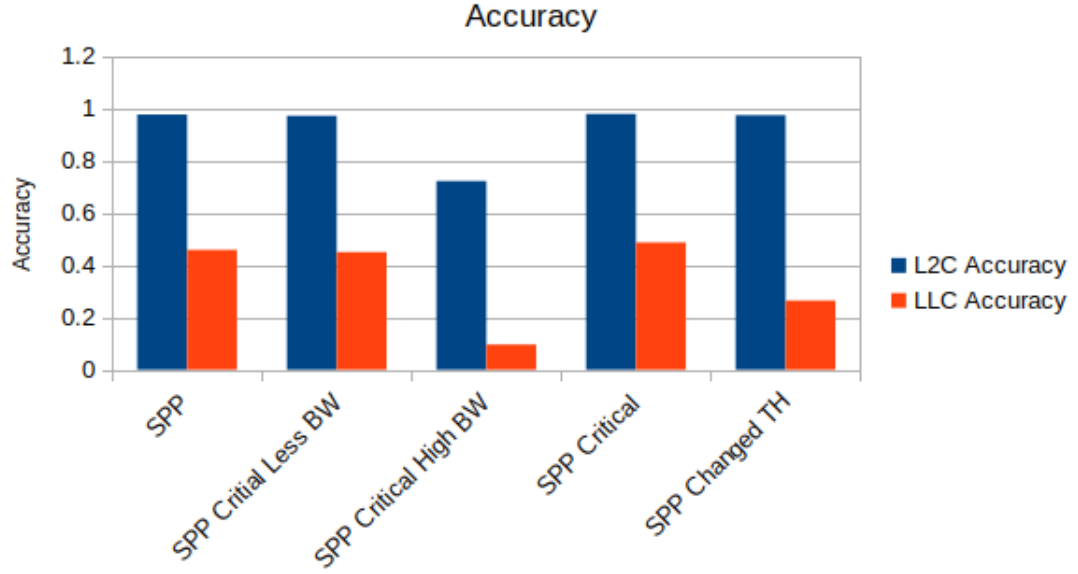


Figure 5.23: Accuracy comparison for quad-core heterogeneous workloads

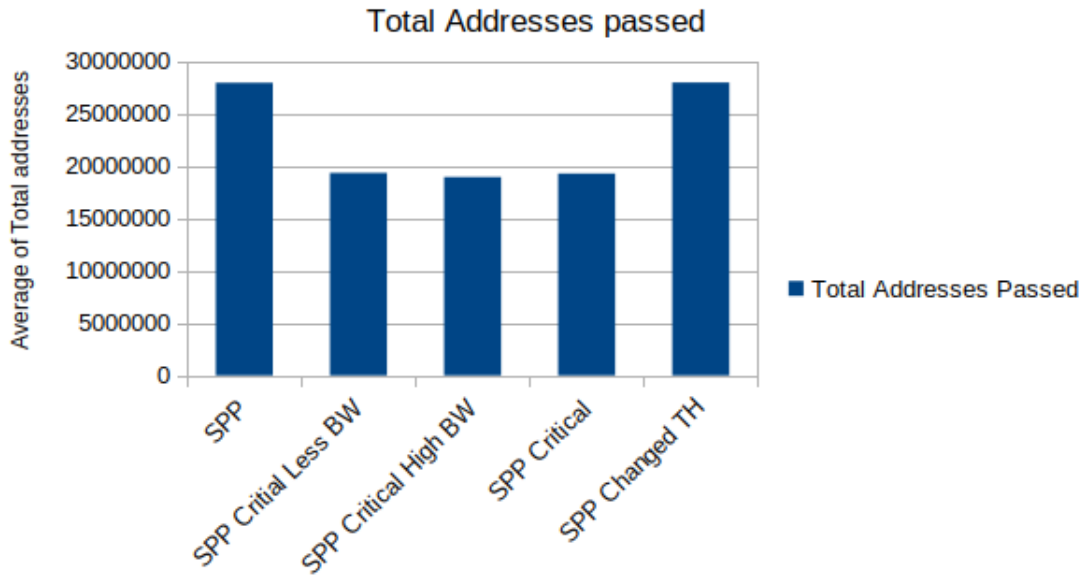


Figure 5.24: Number of addresses input to the prefetcher for quad-core heterogeneous workloads

Prefetch accuracy (Figure 5.23) and the number of addresses input to the prefetcher (Figure 5.24) also behave similarly as in the homogeneous workloads with the high bandwidth proposal having lower accuracy. The major difference between homogeneous and heterogeneous workloads is that SPP generates lots of LLC and DRAM prefetches, while our less bandwidth configuration is able to handle it in the heterogeneous case.

Chapter 6

Conclusions and Future Work

There have been various enhancements and innovations in the prefetching technology since its introduction. Our research has dealt with SPP and applied critical load address-based enhancements to it. We have conducted a thorough evaluation using different configurations to understand the overall behaviour of the prefetcher. We show that with the critical addresses alone, we can get better performance than SPP by properly adjusting the parameters. Finding the optimal set of parameters is a big challenge and we have set these empirically.

Our proposal achieves performance speedup in the single-core configuration with more aggressive prefetching algorithms. In the multi-core configurations, aggressive prefetching results in performance degradation due to pollution and congestion. Our proposal is able to maintain approximately same performance as SPP with the less bandwidth proposal. In most of our evaluation of the less bandwidth proposal, we achieve performance approximately same as SPP with less bandwidth consumption. Even in the configuration where we pass only the critical load addresses to SPP while using its own lookahead mechanism, we are able to maintain the performance of SPP with lower bandwidth consumption.

We observe that managing depth along with pollution holds the key to getting better performance improvement. Our critical load address enhancement to SPP works as an in-built prefetch filter because it generates prefetches only for the critical IPs. The problem of finding the optimal set of parameters or using dynamic methods for this purpose is an open question. We feel that there is a lot of scope for improvement and there are various areas where we can conduct further studies. Some of these are listed in the following.

- We have only done experiments with SPP and seen the effects of critical IPs on SPP only. This can be extended to other prefetchers.
- We have observed that increasing the depth of prefetching increases performance along with increasing cache pollution. To get better results, we need to have some structure to compute

and reduce cache pollution by approximating cache pollution and using it as a feedback while prefetching [10]. Also, advanced prefetch filters can be used to identify and discard the less important prefetches [2].

- Although we are able to limit the performance degradation of multi-programmed workloads using our low bandwidth proposal, further work can be done to have a dynamic combination of high and low bandwidth proposals for achieving better or at least equal performance compared to SPP while optimizing bandwidth consumption.
- As already discussed, more research needs to be conducted for automatically converging to the optimal parameter set at run-time. Some alternate parameters for efficient computation of lookahead can also be thought about.

Bibliography

- [1] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson and Z. Chishti. Path Confidence-based Lookahead Prefetching. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, 2016, pp. 1–12, doi: 10.1109/MICRO.2016.7783763.
- [2] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-based Prefetch Filtering. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*, 2019. DOI:<https://doi.org/10.1145/3307650.3322207>
- [3] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, 2017. DOI:<https://doi.org/10.1145/3037697.3037701>
- [4] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative precomputation: long-range prefetching of delinquent loads. In *Proceedings of the 28th annual international symposium on Computer architecture (ISCA '01)*. Association for Computing Machinery, New York, NY, USA, 14–25. DOI:<https://doi.org/10.1145/379240.379248>
- [5] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. 2001. Dynamic speculative precomputation. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO 34)*. IEEE Computer Society, USA, 306–317.
- [6] R. Manikantan and R. Govindarajan. 2008. Focused prefetching: performance oriented prefetching based on commit stalls. In *Proceedings of the 22nd annual international conference on Supercomputing (ICS '08)*. Association for Computing Machinery, New York, NY, USA, 339–348. DOI:<https://doi.org/10.1145/1375527.1375576>
- [7] Sparsh Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. In *ACM Computing Surveys*, **49**(2): Article 35, November 2016, 35 pages. DOI:<https://doi.org/10.1145/2907071>
- [8] D. A. Jimenez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Monterrey, Nuevo Leon, Mexico, 2001, pp. 197–206, doi: 10.1109/HPCA.2001.903263.
- [9] K. J. Nesbit and J. E. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA '04)*, Madrid, Spain, 2004. doi: 10.1109/HPCA.2004.10030.
- [10] S. Srinath, O. Mutlu, H. Kim and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture*, Scottsdale, AZ, 2007, pp. 63–74, doi: 10.1109/HPCA.2007.346185.
- [11] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks. In *ACM Transactions on Architecture and Code Optimization*, **11**(4): Article 51, January 2015, 22 pages. DOI:<https://doi.org/10.1145/2677956>

- [12] J. E. Smith and G. S. Sohi. The Microarchitecture of Superscalar Processors. In *Proceedings of the IEEE*, **83**(12): 1609–1624, Dec. 1995, doi: 10.1109/5.476078.
- [13] S. Pakalapati and B. Panda. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture*, pp. 118–131, May/June 2020.
- [14] Github Repository of the ChampSim Simulator. <https://github.com/ChampSim/ChampSim>
- [15] SPEC(Standard Performance Evaluation Corporation) benchmark’s Home Page. <https://www.spec.org/cpu2017/>
- [16] Github Repository for the Thesis code. <https://github.com/jatindev2016/ChampSim-master-jatin-thesis>