ABSTRACT

BHAWALKAR, SHUBHAM PRASHANT. Custom Data Prefetchers Evaluated on a Post-Silicon-Microarchitecture-Enabled Superscalar Processor (Under the direction of Dr. Eric Rotenberg).

Conventional CPU cores often suffer from bottlenecks in performance due to cache misses. One way to solve this issue is to use data prefetching to request the data that is likely to be used by the core in the future. Good prefetching, in terms of accuracy, coverage and timeliness leads to high reduction in the load execution latencies and high gains in the core's performance. State-of-the-art general-purpose prefetchers work well to reduce these latencies but suffer from problems such as non-awareness of memory-level parallelism (MLP) and inability to adapt well to complex strided patterns that prevent individual workloads from reaching their maximum IPC potential.

We propose the development of highly customized prefetchers that are tailored to the individual workloads that run on the core. Prefetchers are developed for the highest weighted simpoint of 4 benchmarks of the SPEC CPU 2006 suite. The simpoint runs are profiled to find regions that have delinquent loads with high execution latencies and could benefit from prefetching. The address computation algorithm followed in these regions is mimicked in hardware so that highly accurate prefetches can be generated to effectively prefetch the demand access stream. If complex nested loops exist in the region, we mimic exactly the loop structures, including the number of iterations of the different loop levels, and updates to variables that are involved in the address computation. We use an adaptive mechanism based on performance feedback to establish an appropriate prefetch distance between the demand and prefetch streams, and make the prefetchers MLP-aware for an even reduction in the execution latencies of the delinquent loads.

‘Post-Silicon Microarchitecture’ (PSM) is a novel paradigm that provides the means to synthesize such custom prefetchers on a reconfigurable fabric coupled with the core and enables them to get useful information from the core’s retiring instruction stream via a flexible interface, to help with the prefetching.

We use an in-house cycle-level execute-at-execute superscalar processor simulator to model the prefetchers and run experiments by varying the parameters of the PSM interface, often pairing up the custom prefetcher with another L1D prefetcher or a stream buffer, to obtain more performance boost than either of them could achieve individually. With VLDP as the L2/L3 prefetcher, we observe high gains in performance for the individual benchmarks. The Libquantum benchmark shows a normalized IPC speedup of 3 – 4 (over a baseline with speedup 1) across different PSM configurations and prefetcher combinations, while the LBM benchmark shows a speedup of nearly 4 or more. The Bwaves benchmark shows a normalized IPC speedup of 1.5 – 2 for most configurations. The Leslie3d benchmark shows a speedup of about 1.5 when running standalone and between 1.8 to 2.3 in the presence of another L1D prefetcher or a stream buffer. The development of such custom prefetchers tuned to specific workloads and the high gains in performance they produce greatly strengthens the appeal of using a Post-Silicon Microarchitecture.

Custom Data Prefetchers Evaluated on a Post-Silicon-Microarchitecture-Enabled Superscalar Processor

by
Shubham Bhawalkar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2020

APPROVED BY:

_____	_____
Dr. James Tuck	Dr. Huiyang Zhou

_____
Dr. Eric Rotenberg
Chair of Advisory Committee

BIOGRAPHY

Shubham Bhawalkar was born in Maharashtra, India. He did his schooling in the city of Mumbai, before joining National Institute of Technology, Tiruchirappalli (NIT Trichy) for his undergraduate studies in Electrical and Electronics Engineering. He worked on projects in RTL implementation and verification during his undergraduate days. After graduating from NIT Trichy, Shubham joined NC State University for pursuing a Master of Science degree in Computer Engineering, with a focus on computer architecture. He interned with Samsung Austin R&D Center (SARC), where he worked on exploring hardware prefetching for superscalar cores. Keen to dive deeper into this sub-domain, he joined the research group of Dr. Eric Rotenberg for his MS Thesis to work on developing custom hardware data prefetchers, as a part of a bigger umbrella project called ‘Post-Silicon Microarchitecture’. His first full-time employment is with Nuvia, Inc., as a CPU Design Verification Engineer at Austin, Texas.

ACKNOWLEDGMENTS

I am grateful to several people who have helped in some way or the other during the work for this thesis. Firstly, I would like to extend my heartfelt thanks to Dr. Eric Rotenberg, my thesis supervisor and guide, for providing me with this opportunity and for his constant support and guidance during the thesis. Thanks to him also for helping me set up strong fundamentals in computer architecture through his amazing lectures and coursework which were defining aspects of my graduate studies at NC State and will be the foundation for the career I will pursue in the industry after graduation. Thank you also to Dr. Huiyang Zhou and Dr. James Tuck for being a part of my thesis committee. Thank you to everyone at NC State, especially the faculty of ECE department, for a wonderful two years of Master's filled with classes and coursework which was instrumental in the research that went into this thesis, and will continue to be so for every future endeavor of mine, in the field of computer engineering.

A special thanks goes to my family, to whom I owe everything. Thanks especially to my mother, a constant source of love, motivation and positivity in my life, my father, for instilling discipline and a philosophy to always aim for the best in life, and my younger sister who continues to fill my life with cheerfulness and joy. Thanks to everyone in my extended family and all my friends for their constant love and support.

Many thanks to Chanchal Kumar for introducing me to this novel project of Post-Silicon Microarchitecture and guiding me during the course of my work. Thanks to Anirudh Seshadri for the numerous discussions and debug sessions we had during the course of this work; they were very insightful and enjoyable. Thanks also to all the current and former members of our research group that worked towards the Post-Silicon Microarchitecture project. I would also like to thank everyone I met during my 2019 summer internship with SARC at Austin, especially Knute, Ned

and Tarun, who enabled and inspired me to explore and learn more about this field, especially from the perspective of the industry. My experience during this internship helped greatly with my thesis work.

This thesis was supported in part by NSF grant no.CCF-1823517, and grants from Intel Corporation. Any opinions, findings, and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation or Intel Corporation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Load data prefetching is one of the ways to reduce the latency of delinquent loads (long latency cache-missing loads) by predicting cache lines that will be demanded by the processor in the near future and issuing requests to the lower cache levels or the main memory. This helps unblock the delinquent loads, allowing their dependent instructions to be freed earlier than otherwise.

Typically, prefetchers are developed to be generic, and target multiple workloads. Making a prefetcher general purpose, however, sacrifices performance potential of the individual workloads. For instance, a prefetcher that works well for strided patterns may not work well when the pattern of demand accesses is determined by pointer-chasing or while traversing structures like linked-lists. There have been approaches to make a prefetcher target multiple classes of prefetchers such as IPCP [1] [2], a hybrid collection of different classes of prefetchers. Another way to resolve this high load latency issue is to prefetch using 'pre-execution' paradigms such as runahead execution [3] or slipstream processors [4]. However, we take a different approach here and focus on getting more performance from individual workloads by developing highly customized prefetchers, tailored to the specific workloads.

We develop custom prefetchers by profiling the benchmark's output statistics and the assembly of its source code for regions of interest with delinquent loads that could benefit with load data prefetching. For the purpose of this thesis, the focus has been on complex, regular load access patterns, i.e., the load requests that form a pattern for which the later requests can be predicted computationally using arithmetic and logic operations on the previous demand addresses. Due to the novel microarchitecture paradigm of 'Post-Silicon Microarchitecture' (PSM)

[5][6][7], the developed custom prefetchers can be synthesized on a reconfigurable fabric that is closely coupled with a superscalar core. The prefetcher can then function as if it were a dedicated component of the core and provide IPC improvement.

1.1 Background

General-purpose optimizations such as branch predictors and prefetchers or other ILP techniques may not fetch the best possible performance out of a specific individual application, since they are developed with several workloads in consideration. On a general-purpose core, it is difficult to justify having optimizations with a narrow applicability, which target only a subset of workloads. This is because you would need multiple of those to cover the various classes of applications, which adds to area and power.

Each workload presents its own unique challenges in extracting the best possible performance from it, which may not be met by general-purpose microarchitectural optimizations. This calls for custom microarchitecture development for addressing the specific workloads. As shown in Figure 1.1, PSM consists of a CPU core and a reconfigurable fabric (PSM-RF), such as an FPGA, present on the same chip. Application-specific microarchitecture can be synthesized on the reconfigurable fabric which enables one to squeeze the most performance out of a workload running on the core. A configuration bitstream for the specific workload synthesizes the custom prefetcher on the PSM-RF and configures the ‘PSM Agent’, the interface which facilitates the communication between the core and the PSM-RF.

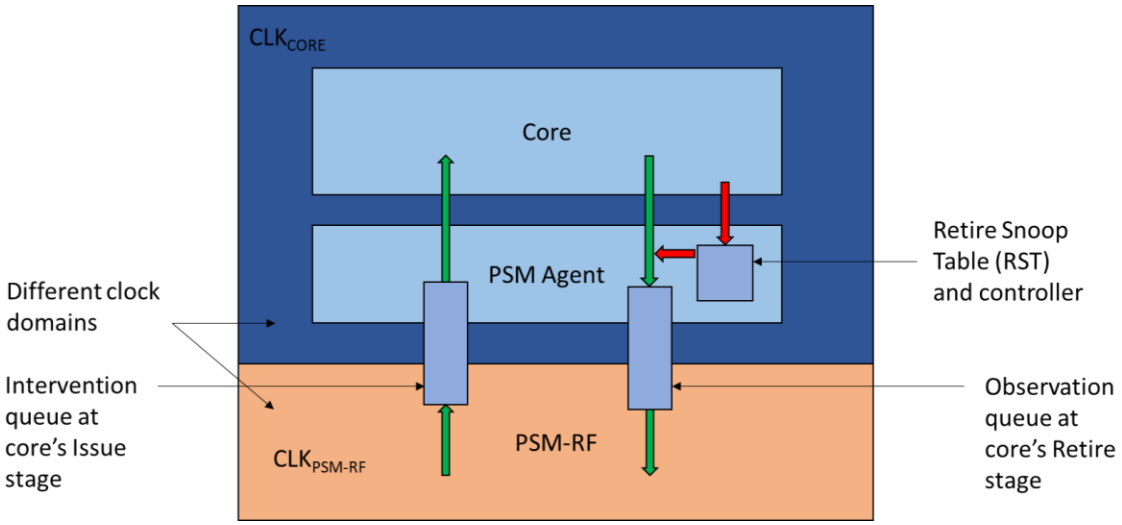


Figure 1.1: High-level block diagram of PSM for prefetching

The PSM Agent includes an 'observation queue' at the retire stage for snooping data from the retiring instruction stream, which is sent to the PSM-RF, and an 'intervention queue' at the core's issue stage for issuing 'prefetch ops' from the fabric to the core, opportunistically. Due to the different clock domain frequencies, the data passed between the fabric and the core is pushed and popped at different frequencies.

PSM prefetcher designs on the PSM-RF need to be decoupled from the core so that they can run ahead of the core's instruction stream. This is because there exists a potential frequency difference between the clock domains of the core and the PSM-RF, and the synthesized design needs to take this into account, in addition to the latency of communication through the queues and the pipelined execution latency on the PSM-RF (which is on a slower clock). Chapter 3 discusses about custom prefetchers that set up the prefetch distance adaptively and make use of this decoupled design to prefetch well ahead of the core's demand access stream.

The PSM Agent also consists of a 'Retire Snoop Table' (RST), which stores the expected PCs of instructions, from which the data that is necessary to operate the prefetcher is to be snooped.

It also determines how packets in the retire observation queue are constructed. When an instruction retires, its PC is compared against the entries in the RST. If there is a match in the RST, the PSM Agent constructs a packet of data and pushes it into the retire observation queue. If it is a load instruction, a corresponding bit is set for this RST entry to indicate to the PSM Agent that it must include the value at the instruction's physical destination register, from the physical register file (PRF), in this packet. The PSM Agent opportunistically shares the PRF access ports of other execution lanes to read values of this load's physical destination register. Note that the PSM Agent has a lower priority for PRF lookup as compared to the core's execution lanes. At the issue stage, the select port of the issue queue is monitored and if there is a bubble in any of the load-store execution lanes, a prefetch op is issued from the head of the intervention queue causing a prefetch request to be sent to the L1 D-cache.

Once the region of interest within a workload is identified, a prefetch generation engine, i.e., an FSM is developed and synthesized on the PSM-RF. The prefetcher uses an adaptive prefetching policy to establish and maintain the distance between the demand access and prefetch access streams. The adaptive mechanism is also part of the logic synthesized on the PSM-RF since the adaptive technique can vary with the workload. Since the PSM-RF logic is inherently slower, establishing a large enough distance between the prefetch and demand streams helps overcome this barrier of clock frequency difference, in addition to the constraints due to the bandwidth and design execution latency. It makes up for the latency that is incurred due to the different frequencies at which data is pushed and popped from the queues.

1.2 Contributions

The work that went into this thesis is a part of a bigger umbrella project on PSM. The focus of this thesis was on use-cases for prefetching, that demonstrate the viability and effectiveness of PSM for workloads that suffer from long-latency cache misses. To that end, custom prefetchers were developed to be integrated into a PSM interface for prefetchers presented in [5]. Specific contributions include:

1. Created a general interface to integrate arbitrary L1 D-cache prefetchers into 721sim, a cycle-level, execution-driven, execute-at-execute superscalar core simulator used in the PSM project.
2. Integrated a state-of-the-art championship prefetcher (IPCP) at L1 D-cache (using the aforementioned prefetcher interface for 721sim) to compare various PSM configurations and prefetcher combinations.
3. Designed custom prefetchers for benchmark simpoints in the CPU SPEC 2006 benchmark suite viz. Libquantum, LBM, Bwaves and Leslie3d and integrated them with the PSM interface. A part of this work was included in a paper published in IEEE Computer Architecture Letters, vol 19 [7].
4. Implemented a stream buffer with configurable parameters, primarily the maximum depth, number of additional stream buffer MHSRs, number of filter entries, and the number of streams. Conducted sweeps to find the best stream buffer configuration for each benchmark, so as to contrast it with the PSM prefetcher, and to show the effect of combining the PSM prefetcher with the stream buffer.
5. Conducted experiments and analyzed results for comparisons and combined operation of the PSM prefetcher and other core prefetchers (viz. the stream buffer, the IPCP prefetcher, and the

next-2-line prefetcher (NLP) integrated previously in [5]), for different configurations of the PSM interface.

1.3 Thesis outline

Chapter 2 discusses related work and introduces the core prefetchers associated with experiments included in the thesis. Chapter 3 presents a discussion on the core L1 D-cache (L1D) prefetcher interface, the implementation of a configurable stream buffer, and the microarchitecture of custom prefetchers and the regions of interest they target in the respective benchmark simpoints [8]. Chapter 4 presents the evaluation methodology and keys for various terms used for the different configurations, and presents the baselines used to calculate IPC speedup. Chapter 5 discusses the experiments performed using the custom prefetchers with PSM, results obtained and further discussion about IPCP, NLP and stream buffers. Chapter 6 summarizes the thesis and discusses future work.

CHAPTER 2

Related Work

2.1 Data Prefetching

Data prefetching is a technique used in order to bridge the gap between the processor's frequency of operation and that of the memory system. The key idea is to bring into the caches the data that is very likely to be accessed in the near future by the processor core. Prefetching works well for programs with predictable memory access patterns. If prefetching has high accuracy and covers a good portion of the demand stream through an adequate number of timely prefetches, it can help reduce cache misses and greatly lower the memory access latency, especially for misses that lead to long-latency accesses to the lower cache levels and the main memory.

Prefetching can be done by software (by compiler/programmer) or hardware. Mowry et al. [9] discuss software-controlled data prefetching wherein instructions for performing the prefetches are inserted into the code by the compiler or the programmer. In case of hardware data prefetching, the prefetcher is a hardware component that observes the memory access pattern, identifies computational or contextual correlations between demand addresses and issues requests for potential future addresses that fit into the identified pattern.

Prefetching requires timeliness to work well. The future data needs to be prefetched early enough for it to be used by the core without significant access latency, but not too early so as to prevent the eviction of useful cache lines (cache pollution). Timeliness is achieved by establishing an appropriate "prefetch distance" between the demand access stream and the prefetch request stream. The prefetch distance corresponds to the difference between the recently demanded

address and the prefetch address that was computed based on the demand address. The address of the prefetch request and the instant at which it is issued both factor into the timeliness.

Adaptive prefetching techniques are often used to adjust the prefetch distance for hardware prefetchers and thus obtain the best prefetching performance. Srinath et al. [10] discuss feedback-directed prefetching, wherein metrics such as the prefetcher's accuracy, timeliness, and cache pollution are dynamically used to adjust the prefetcher's distance and degree (number of prefetches issued per demand access). The simple feedback-based adaptive prefetching techniques employed by the custom prefetchers discussed in this thesis are similar to the policies adopted in this paper.

2.2 Hardware Prefetchers

There are several kinds of prefetchers proposed in academia that target different classes of prefetching depending on indicators such as the delta between consecutive demand addresses or some context which all the demand addresses follow, such as a global or local address history. The simplest prefetcher just prefetches the next cache line following a demand access. For an access to address A, it prefetches the address A+1. This can be expanded to prefetch the next N lines, and a confidence estimation mechanism can be included to control the prefetcher's aggressiveness and monitor its performance.

Stream buffers [18][19] are helpful in reducing compulsory and capacity cache misses. They serve as prefetchers as well as a secondary storage for the L1 D-cache, by prefetching and storing cache lines whose addresses constitute a stream, as demanded by the L1 D-cache accesses. Misses in the L1 D-cache cause allocations in the stream buffer. This helps store lines that are likely to be required by the processor in the near future, without polluting the cache.

Among the state-of-the-art prefetchers is the 'Instruction Pointer Classifier based Prefetching (IPCP)' prefetcher proposed by Pakalapati et al. [1][2], the winner of Data Prefetching Championship 3 [11]. The prefetcher comprises of multiple PC-based prefetchers which target different kinds of access patterns. Instruction PCs of accesses are classified into one of three types – constant stride, complex stride, and stream. Briefly, constant stride refers to an invariant stride that occurs between two consecutive address requests, complex stride is for patterns wherein the strides follow a repeating pattern (e.g., 1, 2, 1, 2, 1, 2, …) and the "stream" prefetching class refers to accesses within a memory region that are cache aligned and can arise from different PCs.

Another state-of-the-art prefetcher of interest is the Variable Length Delta Prefetcher (VLDP) proposed by Shevgoor et al. [12], which acts as the L2/L3 prefetcher, previously integrated into the simulator in [13], and is used for experiments with PSM prefetchers. This prefetcher stores a history of deltas between successive demand misses and uses it to predict future deltas between demand accesses.

The demand accesses of loads can often be dependent on the values of previous loads. Such load-dependent-loads can be challenging to prefetch for, and a custom "load-dependent-load prefetcher" can help provide good performance speedups. [5] presents one such use-case for PSM where a custom prefetcher is developed for the 429.mcf benchmark (from SPEC CPU 2006 suite), which involves pointer-chasing loads.

CHAPTER 3

Baseline and PSM Prefetchers

3.1 L1D Core Prefetcher Interface

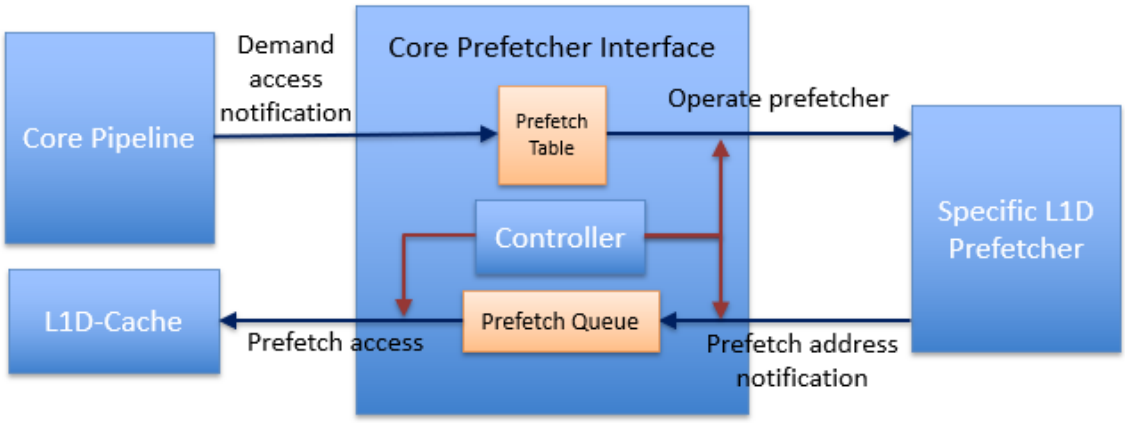


Figure 3.1: Core Prefetcher Interface High Level Diagram

The core L1D prefetcher interface is a prefetch engine to plug arbitrary L1D prefetchers into the core simulator. When a load instruction executes, it uses its load queue index (obtained when it was dispatched into the load queue) to index into the prefetch table and store important information at that entry such as its PC, address, and whether the load access was a hit or a miss. This entry is marked valid, and the information here will be later used to operate the specific L1D prefetcher. This notification of information occurs the first time a load executes. The interface also consists of a prefetch queue into which the prefetch packets are pushed by the specific L1D prefetcher. These packets contain all the information that a cache access may need such as the load's address and instruction PC (if required).

A special index (pointer) ‘pf_op_index’ keeps track of the earliest prefetch table entry that has not yet seen a load access notification. The pointer is incremented for every entry that is processed once it has been marked valid, so that it points to the next entry in the prefetch table.

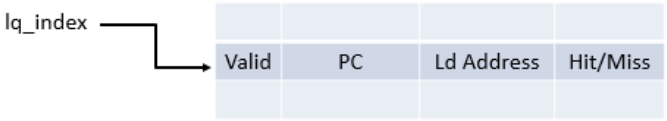


Figure 3.2: Loads access the prefetch table to notify about useful information

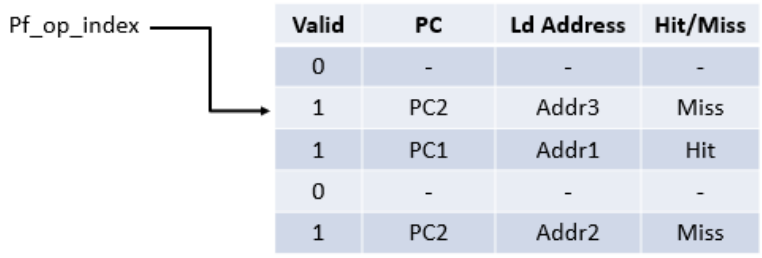
Every core cycle, the prefetch engine looks up the prefetch table using the pf_op_index to find out if the corresponding entry has become valid (i.e., it has received a load access notification). If the entry is valid, the prefetch engine uses this information to operate the specific prefetcher, which then does the necessary decision-making to create and issue prefetch requests that are pushed into the prefetch queue. Until this entry is processed, the prefetch engine does not process the subsequent entries in the prefetch table (even if they are marked valid). This ensures that the specific L1D prefetcher observes the demand accesses of the loads in program order.

Pf_op_index

Valid	PC	Ld Address	Hit/Miss
0	-	-	-
0	-	-	-
1	PC1	Addr1	Hit
0	-	-	-
1	PC2	Addr2	Miss

Figure 3.3: pf_op_index points to the entry where a notification is expected. Due to out-of-order processing of loads, the later entries may be populated before this entry is populated

Once an entry is marked valid, it can be processed, i.e., the information about the demand access at this entry can be made visible to the specific L1D prefetcher for its operation.



Valid	PC	Ld Address	Hit/Miss
0	-	-	-
1	PC2	Addr3	Miss
1	PC1	Addr1	Hit
0	-	-	-
1	PC2	Addr2	Miss

Figure 3.4: Entry pointed to by pf_op_index is now populated due to a notification by a load

Also, the prefetch table is flushed every time the load-store unit suffers from a flush due to mispredictions or load violations.

As long as the prefetch queue has one or more packets to issue prefetches and there is a bubble in any of the load-store lanes, the prefetch engine issues cache access requests every core cycle. For the core configuration used for experiments with 2 load-store lanes, this amounts to at most 2 prefetches every cycle. If no bubble is found in a core cycle, the prefetcher does not issue a prefetch request that cycle, and tries again the next cycle. Further information about the interface can be found in Appendix A.

3.2 Baseline Prefetchers

The baseline core prefetchers used in experimentation for this thesis are:

- **IPCP** - This prefetcher has been briefly described in section 2.2 and is used as a core L1 D-cache (L1D) prefetcher for experiments with the custom PSM prefetchers. The code for the prefetcher has been obtained from the website of the 3rd Data Prefetching Championship [11] and integrated with the 721sim via the L1D core prefetcher interface, which has been discussed in detail in section 3.1.
- **NLP –** The next-2-line prefetcher has been briefly described in section 2.2 and is used as a core L1 D-cache prefetcher for experiments with the custom PSM prefetchers. The code for this prefetcher was integrated into 721sim by Kumar [5].
- **VLDP** – This prefetcher has been briefly described in section 2.2 and is used as a core L2/L3 data prefetcher. The code for this prefetcher was integrated with 721sim by Srinivasan [13].
- **Stream Buffer** – Stream buffers, as introduced in section 2.2, are helpful in reducing capacity and compulsory misses. For purposes of this thesis, a configurable stream buffer was modeled for 721sim. We explain the design of the stream buffer in depth in the remainder of this section.

Stream buffers can have one or more ways (multi-way stream buffers as per [18][19]) which consist of fifo queues along with address generators and other control variables (see Figure 3.5). Each entry of each way consists of a tag, an available bit, and the cache line data. Along with the fifo queues, the stream buffer also uses a hierarchy of two filters – the unit stride filter and non-unit stride filter – to detect a constant stride of 1 or more than 1, respectively. These filters operate

as described in [19]. Also, the stream buffer has been provided with an additional set of MHSRs for tracking its prefetches.

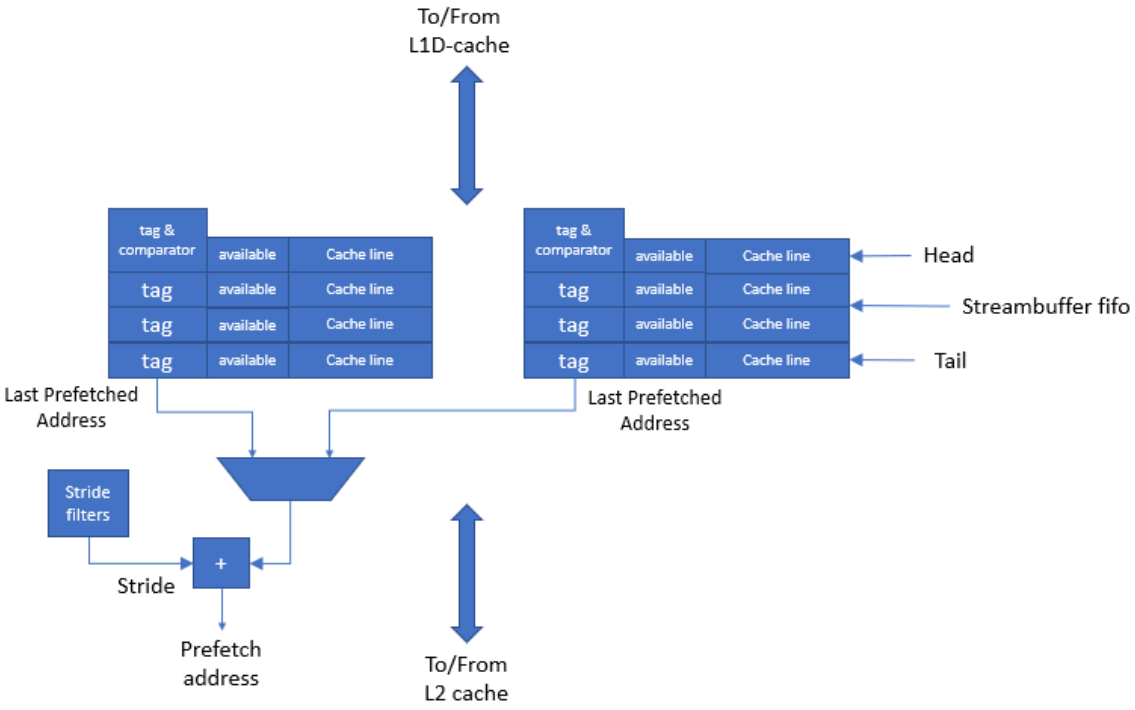


Figure 3.5: A 2-way stream buffer for L1D-cache. Each way is a stream fifo [18][19]

Stream buffers that support the L1 D-cache operate as follows:

- The stream buffer observes the demand access pattern at the L1D-cache and if a stream of consecutive or strided addresses is detected, it records the stride and starts prefetching from the most recent miss address that caused the stream allocation.
- On an L1 D-cache miss, the head of the stream buffer is checked to see if the line has been allocated in the stream buffer.
- If data is available, it is loaded into the L1 D-cache and the head entry is popped from the stream buffer fifo. Note that this happens only if the MHSR occupancy conditions are satisfied, i.e., (1) There is at least one L1 D-cache MHSR available to handle the allocation

of a new cache line from the stream buffer to the L1 D-cache, and (2) The L2 and L3 cache MHSR occupancy thresholds set for the stream buffer are not exceeded in the case of a writeback of the evicted line from L1 D-cache, if it is dirty. Such MHSR occupancy constraints are described in more detail in section 4.2. Further, this causes the next address to be prefetched from the lower level cache. The next address is computed from the address at the tail end of the fifo and the recorded stride. A new cache line will be allocated at the tail end of the fifo queue. When the data is fully loaded in the cache line, the available bit for that entry is set to true.

- If the requested cache line has been allocated at the head of one of the fifos of the stream buffer but the data hasn't been completely loaded (busy MHSR and available bit is reset) then the L1 D-cache must re-attempt reading the stream buffer at a later point for a potential hit.
- If the cache line demanded is not allocated at the head of the fifo queue, the unit-stride filter is checked for an address match. A match indicates the detection of a unit stride and causes a new stream to be allocated, which could potentially evict a previously allocated stream, following the LRU eviction policy. Important information, such as the miss address and the stride, is recorded for prefetching. If no match is found, the non-unit-stride filter is checked for a tag match (a tag would be the higher order bits of the line address). Upon a tag match, an FSM for the matching entry in the filter is operated to determine if a strided pattern in the demand access stream has been detected. A stride detection causes a new stream to be allocated in the stream buffer. Stream fifo entries are allocated cache lines only if free stream buffer MHSRs are available and the MHSR occupancy thresholds are met for the L2 and L3 caches (see section 4.2).

- If the cache line misses in both filters, or if it hits in the non-unit-stride filter but the FSM indicates no stride detection, then the miss handling is handed over to the L1 D-cache. The line will now be allocated in the L1 D-cache directly instead of the stream buffer. Since a stream was not allocated and there was no "hit" in the stream buffer at the head of any of the fifo queues, the stride filters record information regarding the address, as described in [19].
- Every cycle, the stream buffer is checked for entries that did not receive MHSRs previously and aggressively allocates MHSRs to as many fifo entries as is possible, as long as free stream buffer MHSRs are available and the MHSR occupancy thresholds are met for the L2 and L3 caches (see section 4.2).
- Further details regarding the stream buffer and a detailed algorithm can be found in Appendix B.

For the design implemented for this thesis, the stream buffer is configurable with respect to the number of streams, the depth of each stream (number of entries in each fifo queue), sizes of the filters, and the number of additional stream buffer MHSRs. Other configurable parameters, such as the sampling period for adaptive training, initial depth, depth increment, etc., have not been explored for the purpose of this thesis but this can be done in future work.

Also, the stream buffer design is optimistic in the following regards:

- Single cycle hit latency from the L1 to the stream buffer, and back.
- Aggressive issuing of multiple prefetches every cycle, as long as free stream buffer MHSRs are available and the MHSR occupancy thresholds are met for the L2 and L3 caches (see section 4.2)

3.3 Microarchitecture of PSM Custom Prefetchers

3.3.1 Prefetching as a Use-Case for PSM

Due to the decoupled nature of PSM-RF designs, prefetchers which prefetch well ahead of the demand stream can be synthesized on the PSM-RF. The prefetch distance that finally results after the constraints of the bandwidth and pipelined execution latency of the design are considered, is expected to be just right to provide accurate prefetches and cover the demand stream with extremely high accuracy, while also being timely. Figure 3.6 shows how any prefetcher synthesized on the PSM-RF would interface with the core at a high level.

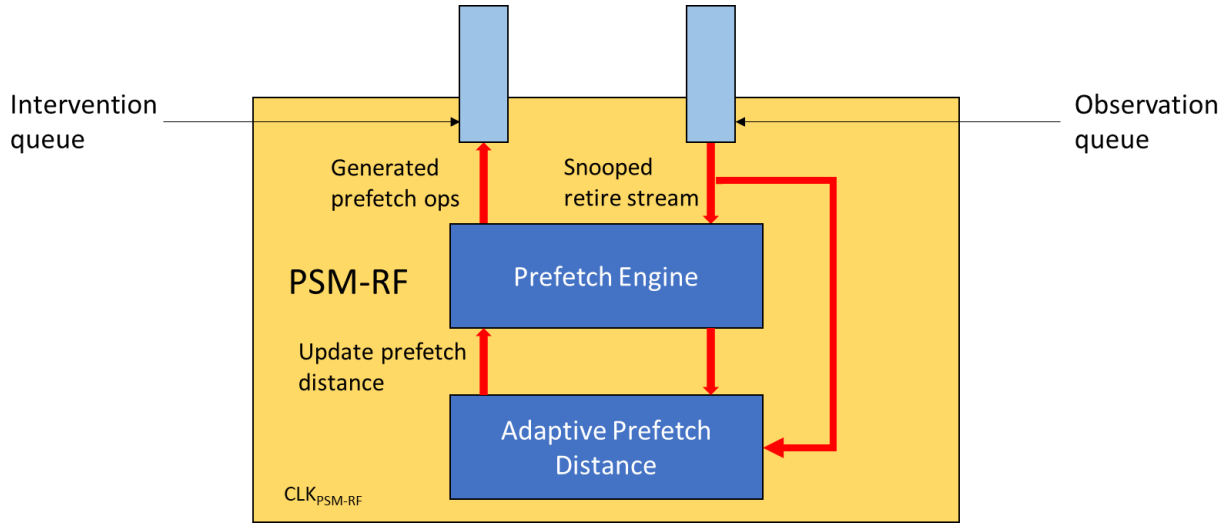


Figure 3.6: High-level block diagram of a custom prefetcher synthesized on PSM-RF

Every custom prefetcher designed follows a common framework for prefetching and interacting with the PSM interface, as follows:

1. Determined by the profile and assembly of the source code of the region of interest, a certain set of PCs are identified as important PCs for the prefetcher's operation. Either values are snooped from the physical destination registers of these instructions or they act

as a starting or stopping point for the PSM prefetching (enable/disable PCs). These PCs are stored in the retire snoop table (RST). If values are snooped, they are added to the packets which are pushed to the observation queue from the core's end.

2. As the instruction flow progresses, just before the region starts, certain PCs are detected by the PSM Agent based on entries in the RST. For these PCs, snooping occurs for their physical destination register values. Typically, these values are the base address (for prefetching) or the number of iterations of the loop(s) in the region of interest, called 'boundl'. In some cases, the 'boundl PC' may also enable and start the prefetching.
3. In the initial phase of the FSM's operation in a particular region of interest, training occurs parallel to the prefetching, to adaptively set the prefetch distance between the demand and prefetch streams. The adaptive policy is different for each workload especially if their regions of interest are different in nature.
4. Prefetching targets the iterations of the loop or nested loop structure, which means that prefetches for more than one load present in an iteration can be bundled and issued together. Generation of prefetching stops when a certain condition is satisfied. The condition could be based on the number of iterations in the target loop, or any other relationship between certain variables inside the loop(s) that signals the end of the loop structure. In some prefetching use-cases, there may be a 'disable' PC which, when encountered in the core's retire stream, sends in a packet down the observation queue to disable the prefetch engine.

3.3.2 Motivation for Custom Prefetchers

Custom prefetchers can extract the best performance out of a workload since they are tailored to the specific regions in that workload that suffer from high cache misses. As opposed to this, general purpose prefetchers target multiple workloads and sometimes also try to include features that could work well across the different classes of demand access patterns. This makes them inefficient at extracting the highest performance potential (e.g., IPC) from individual workloads. This is not feasible when the prefetcher is a dedicated component of the core.

However, due to the novel paradigm of PSM, we can synthesize microarchitectural interventions and optimizations such as branch predictors and prefetchers on the PSM-RF and this hardware is not permanent and can be replaced with another hardware component by re-synthesizing on the PSM-RF. Hence, using custom prefetchers, we can not only get the most out of a certain benchmark, but the concern about having a permanent hardware component with narrow applicability no longer exists.

Further, due to the application-specific nature of the prefetchers, we can incorporate features such as awareness of memory-level parallelism (MLP), to ensure, for instance, that loads of a certain loop iteration are prefetched together. If only some of the loads of the iteration get timely prefetches, then the ones that get very late prefetches become the bottleneck, blocking the head of the reorder buffer for several cycles. This uneven latency reduction renders even the successful prefetches ineffective. General purpose prefetchers typically don’t have such MLP-awareness and therefore suffer from the issue of a shift in the bottleneck between loads of an iteration. Prefetching all the loads together ensures that each load’s latency is evenly reduced and the loop iteration on the whole is sped up in its retirement, thus unblocking the younger instructions.

3.3.3 The Libquantum Custom Prefetcher

The 462.Libquantum benchmark is a part of the SPEC CPU 2006 benchmark suite. There are two regions of interest in this benchmark. The first one is a for-loop within a function 'quantum_toffoli()' which is shown in Figure 3.7. On line 3, there is a delinquent load at 'reg->node[i].state' which follows a constant-stride access pattern.

```
1  for(i=0; i<reg->size; i++)
2  {
3    if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control1))
4    {
5      if(reg->node[i].state & ((MAX_UNSIGNED) 1 << control2))
6      {
7        reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
8      }
9  }
10
```

Figure 3.7: Code snippet for source code of Libquantum (region 1)

Figure 3.8 shows the second region of interest targeted in a function 'quantum_sigma_x' for prefetching using PSM. Similar to the previous code snippet in Figure 3.7, the one in Figure 3.8 shows a delinquent load at 'reg->node[i].state' which follows an access pattern similar to the previous load.

```
1  for(i=0; i<reg->size; i++)
2  {
3    reg->node[i].state ^= ((MAX_UNSIGNED) 1 << target);
4  }
```

Figure 3.8: Code snippet for source code of Libquantum (region 2)

From the assembly of the source code, the address pattern for both loads follows a format base_addr + 8 + 16 * loop_iteration_count. Due to this address pattern, 4 dynamic load instances of consecutive iterations hit in the same cache line of size 64 bytes. As a result, the cache line address pattern will be of the form {A, A, A, A, A + 1, A + 1, A + 1, A + 1, A + 2, A + 2, A + 2,

A + 2, …}, where A is a cache line address. This is a constant-stride pattern (with stride 1) that can be accurately prefetched by an aggressive L2/L3 prefetcher like the VLDP prefetcher used in the baseline. However, VLDP does not seem to be timely enough for the prefetches to be impactful in getting good performance, which is why a profile of VLDP runs for Libquantum show low IPC and high average load execution latency. This makes the loads delinquent. Flowcharts in Figure 3.9, 3.10, and 3.11, explain the key steps that the PSM prefetcher undergoes. Some of the variables used for the purpose of this flowchart are described as follows:

- Boundl: upper bound of loop, corresponds to 'reg->size' in the code snippets shown
- psm_disable_pc: when this instruction is snooped at retire, the PSM prefetcher is disabled
- base_addr: base address from which the base cache line address if obtained for prefetch address calculation
- load_iterations: number of loop iterations with the target load encountered – incremented for every new iteration of the loop, indicated by a certain instruction PC that is snooped. Since 4 successive iterations of the target load cause accesses to the same cache line, the prefetch distance changes when every 4th load iteration retires.
- prev_load_iterations: same as load_iterations but for the previous PSM training quantum (sampling period)
- num_training_cycles: number of cycles for which the training has occurred in a quantum of cycles (in a sampling period)
- training_quantum (sampling period): number of cycles after which prefetch distance is updated based on the number of iterations retired in that quantum

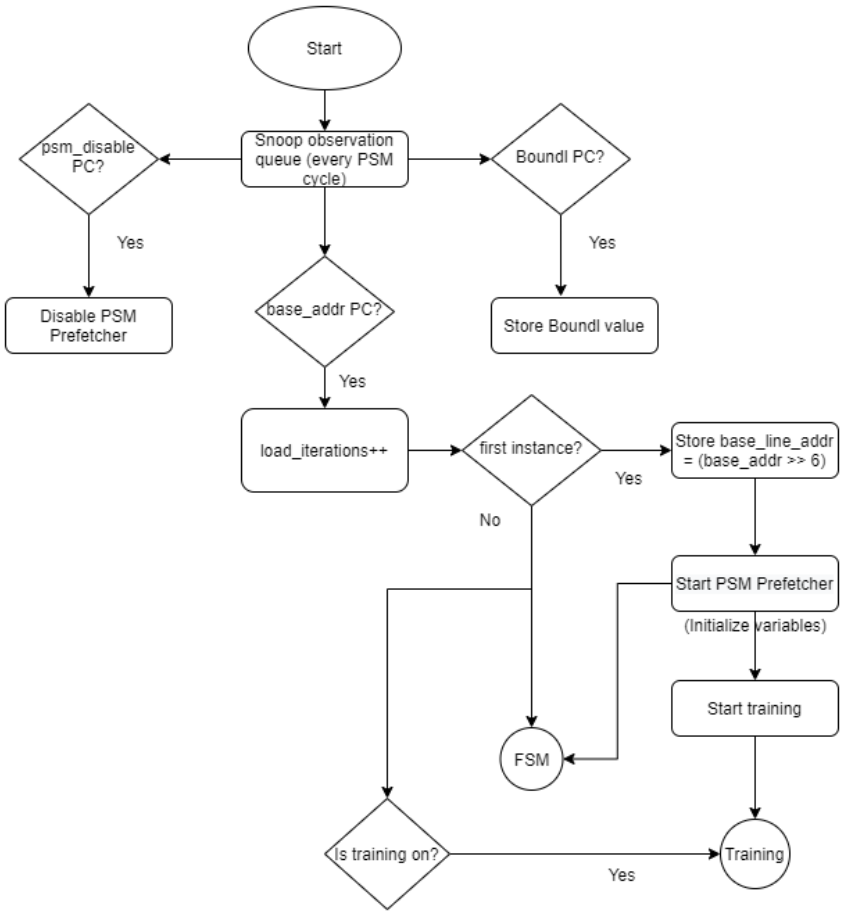


Figure 3.9: Flowchart for decision-making after processing an observation queue packet

For every PSM cycle, the observation queue is checked for packets that correspond to PCs of instructions that guide the PSM prefetcher's behavior. One of these is the 'Boundl PC', a load instruction that provides the number of iterations in the targeted loop. The 'base_addr PC' provides the base address for prefetch address computations. We store the base line address computed from this base address. In the sequence of instructions, this PC marks the beginning of the PSM prefetcher's operation, and provides indication for a new iteration of the loop which has a new dynamic instance of the target load. Also, the prefetch is effectively computed and pushed to the intervention queue only after a new loop iteration is encountered. The way this works is that for

every instance of this base_addr PC retiring, we increment the count for number of iterations (the count is called load_iterations in the flowcharts). We divide this number by 4 since 4 subsequent loads hit in the same cache line. Every 4 iterations, this increases the difference between this number (effectively, the number of cache lines demanded) and the count for prefetches sent (number of cache lines prefetched). This satisfies the prefetch distance condition in Figure 3.10 which compares this difference 'diff' and the adaptively determined prefetch distance. As a result, a new prefetch is issued as long as the intervention queue's space requirements are met.

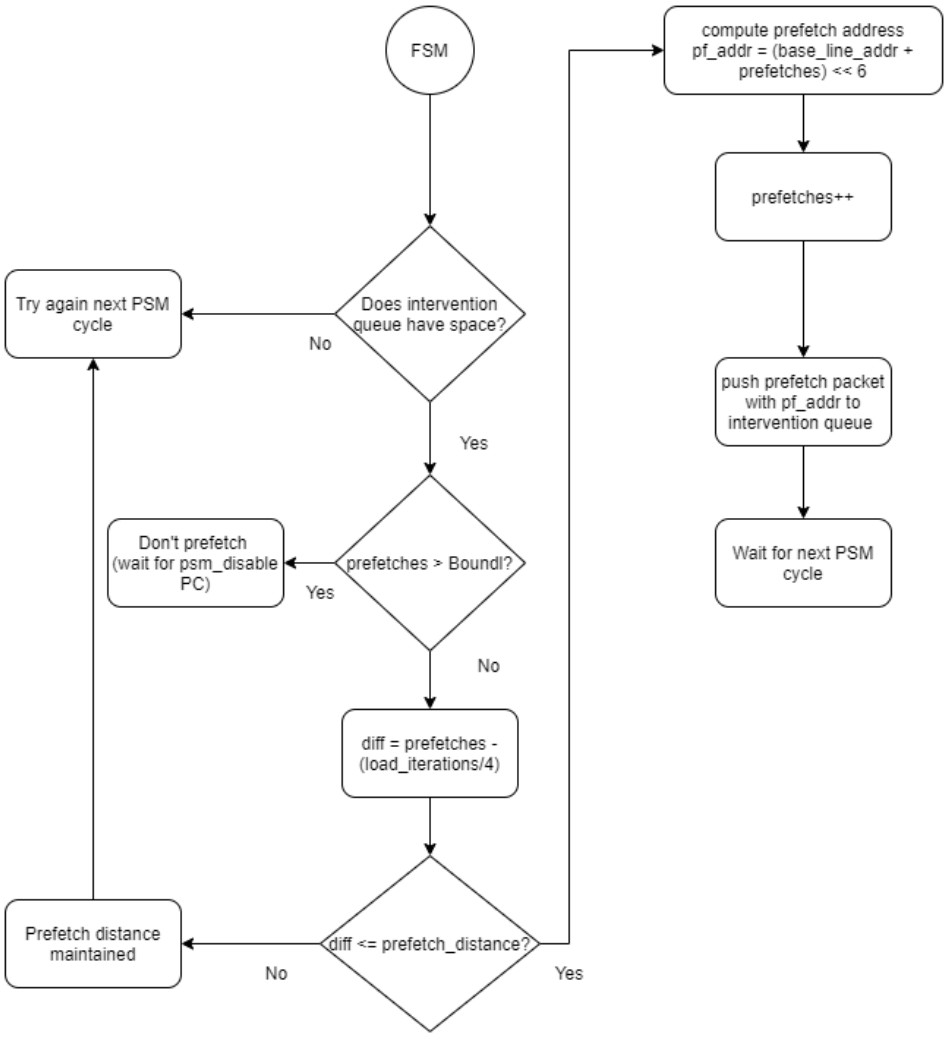


Figure 3.10: Libquantum prefetcher FSM flow

For every PSM cycle that there is space in the intervention queue, a prefetch request is computed and pushed into the queue, as long as it satisfies the prefetch distance condition. This condition ensures that the prefetch stream is ahead of the demand stream by the distance that was selected adaptively during training. If enough number of iterations have not been encountered, additional prefetches are not computed until this requirement is met. The prefetching continues until the the psm_disable_pc is observed from the observation queue. This effectively takes care of prefetching only those addresses that would exist in the demand stream at any point in the program, and no more. A code snippet in Figure 3.11 shows the point where address is computed and pushed into the intervention queue. The ‘delay queue’ shown here is a program construct that effectively models the pipelined execution latency of the custom design synthesized on the PSM-RF.

```
if((prefetch_iter_count_-(num_target_loads_ret_ / 4)) <= prefetch_distance_){
    uint64_t addr = (base_line_addr + prefetch_iter_count_) << 6; // libquantum
    ++prefetch_iter_count_;
    if(p2c_pipe_delay_size_ > 0){
        assert(p2c_pipe_delay_queue_.size() <  p2c_pipe_delay_size_);
        p2c_pipe_delay_queue_.push_back(std::pair<bool,PSM2CoreQueueEntry>(true, PSM2CoreQueueEntry(COMMAND::PREFETCH, addr, 0)));
    }
```

Figure 3.11: Code snippet for address computation and pushing to intervention queue

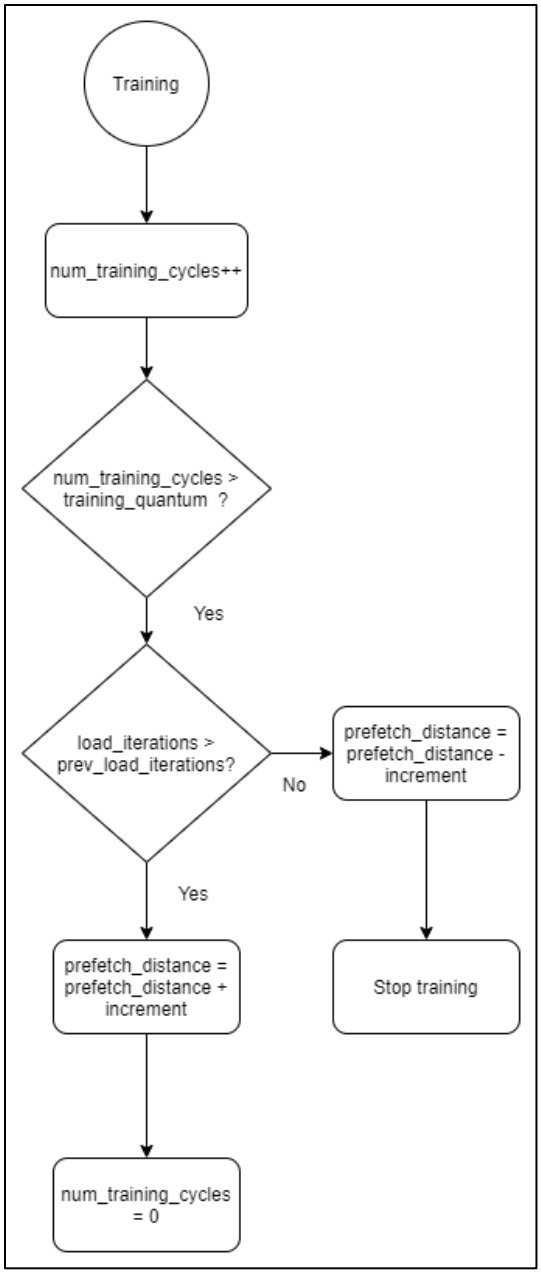


Figure 3.12: Training

Adaptive training occurs parallelly with the issuing of prefetch requests. The first instance of the ‘base_addr PC’ retiring also marks the starting of the training phase. This phase is divided into ‘quantums’, similar to sampling periods, where the number of iterations retiring in that quantum are compared with those retiring in the previous quantum. If more iterations retired this

quantum, then we increase the prefetch distance to see if a greater distance provides a better performance, potentially through more timeliness of the prefetches. The best distance is found when the peak is reached such that a further increment in the prefetch distance causes the number of iterations retiring in a quantum to decrease, potentially due to prefetches that are too early and cause cache pollution. After two successive quantums of decreasing performance (two for hysteresis) the prefetch distance is reverted to its value in the previous training quantum as the best distance, and training ends. A code snippet in Figure 3.13 shows this as written for 721sim. The 'decr_dist_next' is a variable added for hysteresis in the decision making.

```
// adaptive training
if(training_){
    ++num_train_cycles_;
    if(num_train_cycles_ > num_cycles_quantum_){
        if(num_iter_retired_train_ > prev_num_iter_retired_train_){
            prev_num_iter_retired_train_ = num_iter_retired_train_;
            prefetch_distance_ += PREFETCH_INCREMENT;
            prefetch_iter_count_ += PREFETCH_INCREMENT;
            decr_dist_next = false;
        }
        else if(!decr_dist_next){
            decr_dist_next = true;
        }
        else{
            prefetch_distance_ -= PREFETCH_INCREMENT;
            std::cout << "Final prefetch distance: " << prefetch_distance_ << std::endl;
            fprintf(proc_->stats_log, "Final prefetch distance: %d\n", prefetch_distance_);
            training_ = false;
            decr_dist_next = false;
        }
        num_train_cycles_ = 0;
        num_iter_retired_train_ = 0;
    }
}
```

Figure 3.13: Code snippet for adaptive training for Libquantum

3.3.4 The LBM Custom Prefetcher

The 470.LBM benchmark is part of the SPEC CPU 2006 benchmark suite. There is one region of interest in the LBM benchmark that benefits greatly with custom prefetching using PSM and achieves results that are very close to the results obtained for a perfect data cache configuration. The region of interest is in a function 'LBM_performStreamCollide' that has 6 delinquent loads which follow a constant stride pattern, similar to Libquantum, but demand different cache lines. The loads need to be prefetched as a cluster to prevent any one of the loads from becoming the bottleneck. The VLDP prefetcher seemingly fails to exploit this memory-level parallelism (MLP) but it is captured by the custom prefetcher. This is highlighted in the results for LBM prefetcher in section 5.2.

The key aspects of this prefetcher's operation related to the PSM interface, including its interaction with the queues for snooping and pushing prefetches, are very similar to that of Libquantum prefetcher discussed earlier. There are no PCs providing 'boundl' or 'psm_disable_pc' since the code flow for 100M instructions stops before the loop completes. As a result, the PSM prefetcher's operation is started when a 'base_addr' PC is detected for the first time and the operation ends when the simulator stops. Training progresses similarly to Libquantum, in quantums of training cycles (sampling periods) to reach the best prefetch distance adaptively through performance feedback in the form of number of loop iterations retiring in a quantum.

The key difference for this prefetcher design as compared to the Libquantum prefetcher is the prefetch address computation logic as shown in the following code snippets from the prefetcher's design in Figure 3.14.

```
1   uint64_t PrefetchGenerationEngine::get_pf_addr(){
2       uint64_t addr;
3       load_id_++;
4       if(load_id_ == 7)
5           load_id_ = 1;
6
7       switch(load_id_){
8           case 1: {
9                       addr = base_addr + 152 + 160*prefetch_iter_count_;
10                      break;
11                  }
12          case 2: {
13                      addr = base_addr + 16 + 160*prefetch_iter_count_;
14                      break;
15                  }
16          case 3: {
17                      addr = base_addr + 48 + 160*prefetch_iter_count_;
18                      break;
19                  }
20          case 4: {
21                      addr = base_addr + 72 + 160*prefetch_iter_count_;
22                      break;
23                  }
24          case 5: {
25                      addr = base_addr + 112 + 160*prefetch_iter_count_;
26                      break;
27                  }
28          case 6: {
29                      addr = base_addr + 144 + 160*prefetch_iter_count_;
30                      break;
31          default: {
32                       printf("Something wrong with pf addr gen");
33                       assert(0);
34                   }
35      }
36
37      if(load_id_ == 6)
38          ++prefetch_iter_count_;
39
40      return addr;
41  }
42
```

Figure 3.14: Code snippet for LBM prefetch address computation

The 6 addresses are generated once for each cluster of 6 loads that are prefetched together. For each of these loads, there is a corresponding call to the address computation function ‘get_pf_addr()’ and once all prefetch addresses for all the 6 loads have been computed, the prefetch count is incremented as shown in the code in Figure 3.14 on line 38. Note that these are byte

addresses being computed. This is followed by pushing all 6 prefetches together to the intervention queue. In this way, the memory level parallelism is exploited by prefetching all the loads together, evenly.

3.3.5 The Bwaves Custom Prefetcher

The 410.Bwaves benchmark is a part of the CPU SPEC 2006 benchmark suite. The region of interest in this benchmark is in a function ‘mat_times_vec’, shown in the code snippet in Figure 3.15.

```
      do k=1,nz
         km1=mod(k+nz-2,nz)+1
         kp1=mod(k,nz)+1
         do j=1,ny
            jm1=mod(j+ny-2,ny)+1
            jp1=mod(j,ny)+1
            do i=1,nx
               im1=mod(i+nx-2,nx)+1
               ip1=mod(i,nx)+1
               do l=1,nb
                  y(l,i,j,k)=0.0d0
                  do m=1,nb
                     y(l,i,j,k)=y(l,i,j,k)+
1                    a(l,m,i,j,k)*x(m,i,j,k)+
2                    axp(l,m,i,j,k)*x(m,ip1,j,k)+
3                    ayp(l,m,i,j,k)*x(m,i,jp1,k)+
4                    azp(l,m,i,j,k)*x(m,i,j,kp1)+
5                    axm(l,m,i,j,k)*x(m,im1,j,k)+
6                    aym(l,m,i,j,k)*x(m,i,jm1,k)+
7                    azm(l,m,i,j,k)*x(m,i,j,km1)
                  enddo
               enddo
            enddo
         enddo
      enddo
```

Figure 3.15: Code snippet for region of interest in Bwaves benchmark

It is a 5-level nested loop structure with several loads. These loads have a demand access pattern that is a function of variables from different loop levels. For example, at the line marked

by ‘1’ we see an array access a(l,m,i,j,k) where l-k are index variables that are updated at different levels of the loops. Among these loads, there are seven loads that have a high latency, as indicated by the profile of the benchmark. These loads are candidates for prefetching. To get accurate prefetches for these loads with the multi-strided access pattern, we need a highly benchmark-specific customized prefetcher, tailored to the benchmark’s region of interest.

The prefetcher’s interaction with the PSM interface (observation and intervention queues) is similar to that of previously discussed prefetchers for LBM and Libquantum. There is a trigger instruction PC that starts off the PSM prefetcher when it is snooped from the observation queue. The simpoint starts in the middle of the loop structure and a 100M run ends without finishing the outermost loop. As a result, the custom prefetcher has been designed to detect the trigger PC, start prefetching, and stop as the simulator exits. Another instruction PC, when snooped, helps keep track of the number of iterations encountered so far, and thus help in prefetching to maintain the prefetch distance. There are 7 loads targeted so that we issue 7 prefetches as a cluster for each iteration of the loop that is encountered.

The optimal prefetch distance is found adaptively by training in quantums of cycles (sampling periods) just like Libquantum as discussed earlier. Additionally, the Bwaves prefetcher also retrains for every 100,000 loop iterations that retire. This is one way that the adaptive prefetching is made more robust.

Prefetching starts once the trigger PC is observed and continues parallelly with training. Once training ends, a prefetch-drop policy is used to maintain the prefetch distance throughout the course of the program. If at any point the intervention queue does not have space to accommodate additional prefetch requests, the prefetches produced due to a new iteration are dropped. Since it is a cluster of prefetches that are either pushed to the queue or dropped all at once, the prefetching

is evenly done, and distance is maintained across all loads and their respective prefetches. Further, dropping prefetches as a cluster ensures that the latency reduction for the loads is even, thus exploiting the memory-level parallelism (MLP).

The custom prefetcher is modeled such that it very closely resembles the assembly of the source code of the Bwaves benchmark. This mimicking of the assembly is necessary to capture the load address computation accurately. The following code snippets, in Figures 3.16 and 3.17, show portions of the simulator code for the Bwaves custom prefetcher FSM.

```
int64_t a5 = i1_a + i3_b + 8 + i2;
string s = "i1";

temp_line_addr_[0] = s10 + a5;  // 10ee4
temp_line_addr_[1] = s7 + a5;   // 10ef8
temp_line_addr_[2] = s6 + a5;   // 10f08
temp_line_addr_[3] = s5 + a5;   // 10f1c
temp_line_addr_[4] = s4 + a5;   // 10f2c
temp_line_addr_[5] = s8 + a5;   // 10f40
temp_line_addr_[6] = s9 + a5;   // 10f54
```

Figure 3.16: Code snippet for Bwaves custom prefetcher - address computation

There are 7 loads targeted for prefetching as shown in Figure 3.16. Each of the addresses is dependent on variables assigned or updated at different loop levels. Each address is computed from a constant (s4,s5,…,s10) and a variable a5 which is itself dependent on a few other variables updated at different loop levels.

To demonstrate the dependencies across loop levels, consider the code snippet in Figure 3.17. As shown in Figure 3.17, in the highlighted line 65, the variable i3_b is re-initialized in the 4th level of the loop. The variable is a function of two constants and a variable i4_b which itself is re-initialized in the 5th level of the loop to i5_b, and i5_b is also updated at the 5th level. As

shown in Figure 3.16, this variable i3_b is used for address computation. Similarly, all other variables are computed throughout the loop levels, finally contributing to the load addresses.

```
46                      i5_a += i5_a_inc;
47                      i5++;
48                      i5_b += i5_b_inc;
49
50
51
52                      if(i5 >= bound5){
53                          //pf_instance_over = true;
54                          // shouldn't reach here
55                          assert(0);
56                      }
57
58                      i4 = 1;
59                      i4_a = i5_a + i4_a_inc;
60                      i4_b = i5_b;
61                  }
62
63                  i3 = 0;
64                  i3_a = i3_a01 + (i3_a02 + i4_a) << 3;
65                  i3_b = (i3_b01 + i3_b02 + i4_b) << 3;
66              }
67
68              i2 = 0;
69              bound1 = bound1_0 + i3_a;
70          }
71
72          i1 = i3_a;
73          i1_a = 0;
74      }
75
```

Figure 3.17: Code snippet for Bwaves custom prefetcher – updation of variables used in address computation

3.3.6. The Leslie3d Custom Prefetcher

The 437.Leslie3d benchmark is a part of the SPEC CPU 2006 benchmark suite. There are 3 regions in the source code of Leslie3d that were targeted, considering the highest weighted simpoint. Two of those regions lie inside a function ‘FLUXJ_’ and one inside function ‘EXTRAPK_’. Each region is a nested-loop structure with two to four levels. A profile of the benchmark simpoint led to identifying the long-latency cache-missing loads and the subsequent development of a custom prefetcher that nearly mimics the assembly for these 3 regions so as to obtain accurate prefetches. Each region has its own independent FSM for computing the addresses.

The custom prefetcher was built like the one for Bwaves in section 3.3.5. The adaptive training mechanism uses multiple candidates for prefetch distances and chooses one at the end of the training phase. The cluster prefetch and drop policy helps maintain the prefetch distance throughout the program. Each region has its own starting PC to invoke the corresponding prefetcher FSM. The prefetcher FSM is disabled as the last prefetch address corresponding to the end of the loop is issued, similar to Libquantum. The end point is naturally arrived at, since the FSM mimics the assembly and once all the loop iteration bounds are reached, the prefetching stops.

Similar to the Bwaves custom prefetcher, the variables involved in address computation have updates and re-initializations occurring over different loop levels. The code snippets in Figure 3.18 and Figure 3.19 show the FSM for one of the 3 ROIs in the Leslie3d simpoint, which has 4 loop levels.

```
1   // PSM prefetch generation fsm
2   void PrefetchGenerationEngine::gen_pf_packet_reg1(){
3
4       a5 = a4 + a6;
5       a7 = a5 + a0;
6
7       temp_line_addr_[0] = a7 - 8; // load 2 i.e 183d4
8       temp_line_addr_[1] = a4 - 8; // load 1 i.e. 183d8
9
10      a5 += a1;
11
12      temp_line_addr_[2] = a5 - 8; // load 3 i.e. 183e4 772
13
14
15      a3++;
16      a4 += 8;
17
18      if(a3 == t0){    // loop i1 branch
19
20          t1++;
21          a0 += t5;
22
23          a6 += s0;
24
25          if(t1 == s3){  // loop i2 branch
26              s4++;
27              s9 += sp_24;
28              s6 += s11;
```

Figure 3.18: Code snippet for Leslie3d custom prefetcher: prefetch address computation and updation of variables involved in the computation

The lines 7 – 12 in Figure 3.18 show the addresses being generated. The variables used for this address generation are dependent on other variables at different loop levels. With such a multi-strided access pattern, a good prefetcher would struggle to provide the best performance due to the complex address generation and the existence of a cluster of cache missing loads (here, 3 loads) that need to be prefetched together to exploit MLP for higher performance.

```
30                  if(s4 == sp_32){  // loop i3 branch
31
32                      sp_72--;
33                      sp_80 += sp_112;
34
35
36                      s10 += sp_48;
37                      if(sp_72 == 0){ // loop i4 branch
38                          // end of loop
39
40                          is_fsm_reg1_done = true;
41
42                      }
43                      s6 = s10 + sp_96;
44                      s9 = sp_80;
45                      s4 = sp_120;
46                  }
47
48                  a6 = (-(s10 + s11 * s4)) << 3;
49
50
51                  t3 = (s10 + s11*s4 - s6) << 3;
52
53
54                  a0 = s9;
55
56                  t4 = (s10 + (s4 + sp_40) * s11 - s6) << 3;
57
58                  t1 = 0;
59              }
60              a3 = -1;
61
62              a4 = a0 + t3 + s5;
63              a1 = a0 + t4;
64
65          }
66
67      }
68
```

Figure 3.19: Code snippet for Leslie3d custom prefetcher - updation of variables used for address computation

CHAPTER 4

Evaluation Methodology

4.1 Configurations for Baseline Core and Prefetchers

All experiments are performed using 721sim, a cycle-level, execution-driven, execute-at-execute, superscalar processor simulator. The baseline core configuration is presented in Table 4.1. Two different prefetching baselines are used, without and with VLDP L2/L3 prefetcher, as described in Table 4.2. In the plots presented in future sections, the baseline is always assumed to be at value 1 (i.e., speedup 1). Also, the highest weighted simpoint of the respective SPEC benchmark, compiled to the RISC-V ISA [14], is used for the experiments. The key for various stream buffer parameters is shown in Table 4.3. The key for various PSM parameters is shown in Table 4.4.

Table 4.1: Baseline core configuration for experiments

Core	Superscalar out-of-order, 10-stage from Fetch to Retire; Fetch/Retire width: 4 instr/cycle Dispatch/Issue width: 8 instr/cycle ALUs: 4 Simple, 2 FP-Complex and 2 LS ROB: 224 entries, IQ: 100 entries, LQ-SQ: 72 entries each, PRF: 288 entries
Branch Predictor	Predictor: 64KB TAGE-SC-L predictor [15]; BTB: 4k entries, 4-way set-associative RAS: 32 entries
Memory Hierarchy	L1I and L1D: split, 32KB each, 8-way set-associative, 1 cycle access latency (4-cycle load-to-use latency), 32 MHSRs each L2: Unified, 256KB, 8-way set-associative, 12-cycle access latency, 64 MHSRs L3: Unified, 8 MB, 16-way set-associative, 42-cycle access latency, 128 MHSRs DRAM: 250-cycle access latency

Table 4.2: Prefetching baselines used for experiments. Note that a given baseline is always at speedup 1 in the plots.

Prefetching Baseline	Core Modifications
No_pf_baseline	-
Vldp16_baseline	L2/L3 prefetcher: VLDP (5.5 Kb) [+]

[+] MHSR occupancy threshold of 16 out of 64 L2 MHSRs and 96 out of 128 L3 MHSRs. Section 4.2 elaborates on the MHSR occupancy thresholds for prefetchers.

Table 4.3: Key for stream buffer configurations

Parameter	**Description**
mdM	Maximum depth of the stream buffer
mhsrM	Number of additional MHSRs provided to the stream buffer
stS	Number of streams
filtF	Number of filter entries for the unit-stride and non-unit stride filters

Table 4.4: Key for PSM configurations

Parameter	**Description**
clkC	C is the ratio of the frequencies of PSM-RF and the core, $C = \frac{f(CLK_{CORE})}{f(CLK_{PSM-RF})}$
wW	W is the superscalar width of PSM; PSM-RF can push/pop W payload packets to/from the interface queues every $CLK_{PSM\text{-}RF}$ cycle
delayD	D is the execution latency on the PSM-RF in $CLK_{PSM\text{-}RF}$ cycles (e.g., for a clk4_w4 config and delay4, the execution latency would be 4 $CLK_{PSM\text{-}RF}$ cycles or 16 CLK_{CORE} cycles)
queueQ	Q is the size of observation and intervention queues

4.2 MHSR Occupancy Thresholds for Prefetchers

All core prefetchers viz. IPCP, NLP, VLDP and the stream buffer are subjected to MHSR constraints to assign higher priority to demand misses over prefetch misses and to avoid delaying forward progress in the core (the performance could tank if too many MHSRs are occupied with prefetches, possibly inaccurate ones, taking resources away from demand misses). Unlike the general purpose prefetchers, the PSM prefetcher generates accurate prefetches that mimic the demand stream, and hence does not need this restriction. Table 4.5 shows these limits. In sections 5.5.2 and 5.5.3 we relax some of these restrictions to investigate the effect of providing this freedom to core prefetchers on IPC, thereby disregarding the MHSR occupancy.

Table 4.5. MHSR Occupancy Thresholds for core prefetchers

Prefetcher	**Prefetch into the L1 D-cache only if**	**Prefetch into the L2 cache only if**	**Prefetch into the L3 cache only if**
VLDP	N/A (Prefetches into L2 and L3)	Number of L2 MHSRs occupied < 16 Number of L3 MHSRs occupied < 96	Number of L3 MHSRs occupied < 96
Stream Buffer	N/A (Prefetches into L2 only)	Number of L2 MHSRs occupied < 48 Number of L3 MHSRs occupied < 96	Number of L3 MHSRs occupied < 96
NLP	Number of L1 D-cache MHSRs occupied < 16	N/A	N/A
IPCP	Number of L1 D-cache MHSRs occupied < 16	N/A	N/A

For the VLDP prefetcher, the limit on L2 MHSR occupancy is 16 out of total 64 L2 MHSRs available, i.e., prefetch only if the number of L2 MHSRs occupied is less than 16. The L3 MHSR occupancy threshold for VLDP is 96 out of 128 L3 MHSRs, which is checked for L2 and L3 prefetches, both. For example, to prefetch into L2, the number of L2 MSHRs occupied currently should be less than 16 and the number of L3 MHSRs occupied currently should be less than 96. This check is performed for both cache levels because an L2 prefetch could miss in both, the L2 and L3 caches, causing MHSR allocation at the L2 and L3 levels. To prefetch into the L3 cache,

only the L3 cache MHSR occupancy is checked, i.e., we prefetch into L3 only if the number of L3 MHSRs occupied is less than 96.

For the L1D prefetchers that are used in the experiments herein viz. IPCP and NLP, the MSHR occupancy threshold is 16 out of 32 L1 D-cache MHSRs, i.e., prefetch into L1 D-cache only if the number of occupied L1 D-cache MHSRs is less than 16.

For the stream buffer configurations, the MHSR occupancy threshold for any prefetch access to L2 is 48 out of 64 L2 MHSRs and 96 out of 128 L3 MHSRs. This check is performed every time a prefetch is sent to L2 for the tail entries of the stream fifos and also whenever a cache line is to be loaded from the stream buffer into the L1 D-cache, on a stream buffer hit. In the latter case, there is possibility of writeback to L2 cache if the evicted L1 cache line is found to be dirty.

4.3 Stream Buffer Configurations

The stream buffer that has been modeled for 721sim has configurable parameters. We find the best configuration in terms of number of streams, maximum depth, maximum number of dedicated MHSRs available, and the size of filters for the stream buffer. For Libquantum, this results in the plots shown in Figures 4.1 through 4.3.

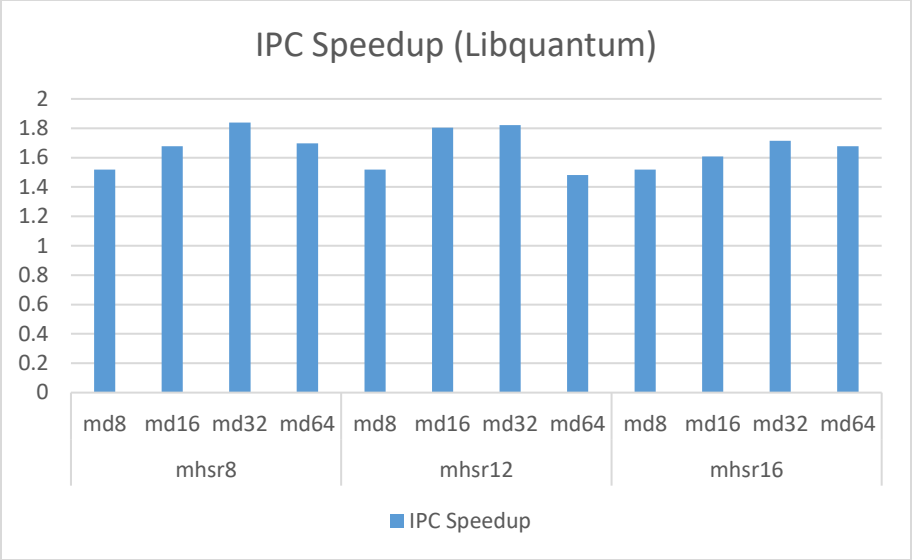


Figure 4.1: IPC Speedup for different maximum depths 'mdD' and stream buffer MHSRs 'mhsrM' ' for 1 stream (filt16). The best config for single stream (st1) is md32 and mhsr8 with a speedup of more than 1.8. The baseline used is Vldp16_baseline.

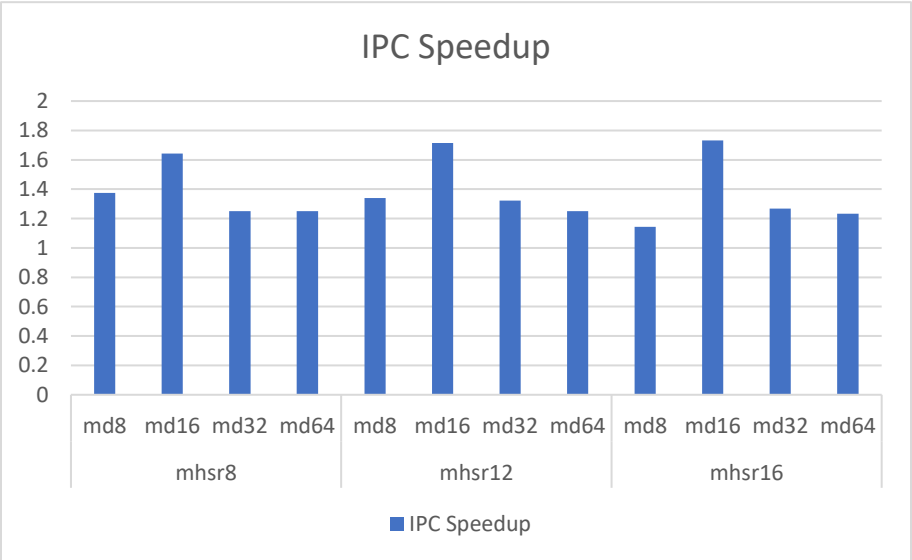


Figure 4.2: IPC Speedup for different maximum depths 'mdD' and stream buffer MSHRs 'mhsrM' ' for 2 streams (filt16). The best config for 2 streams (st2) is md16, mhsr16 with a speedup of about 1.73. The baseline used is Vldp16_baseline.

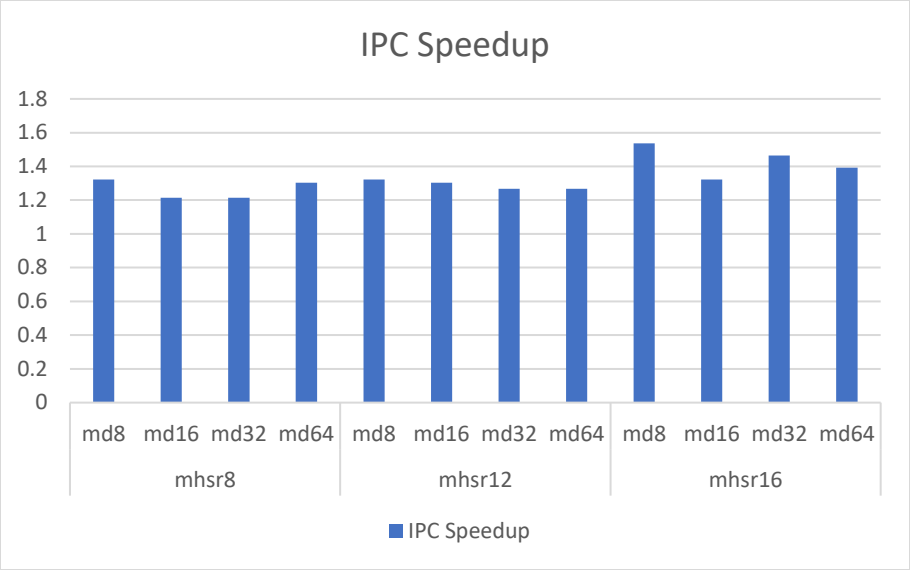


Figure 4.3: IPC Speedup for different maximum depths 'mdD' and stream buffer MHSRs 'mhsrM' ' for 4 streams (filt16). The best config for 2 streams (st2) is md8 and mhsr16, with a speedup close to 1.5. The baseline used is Vldp16_baseline.

From all three plots, it is evident that, md32, mhsr8, and st1, is the best configuration for the stream buffer (assuming a filter size of 16). To confirm that a filter size of 16 is sufficient, we plot the IPC speedup for different filter sizes in Figure 4.4. Any filter size greater than or equal to 8 should work, since they produce identical IPC and cycle count.

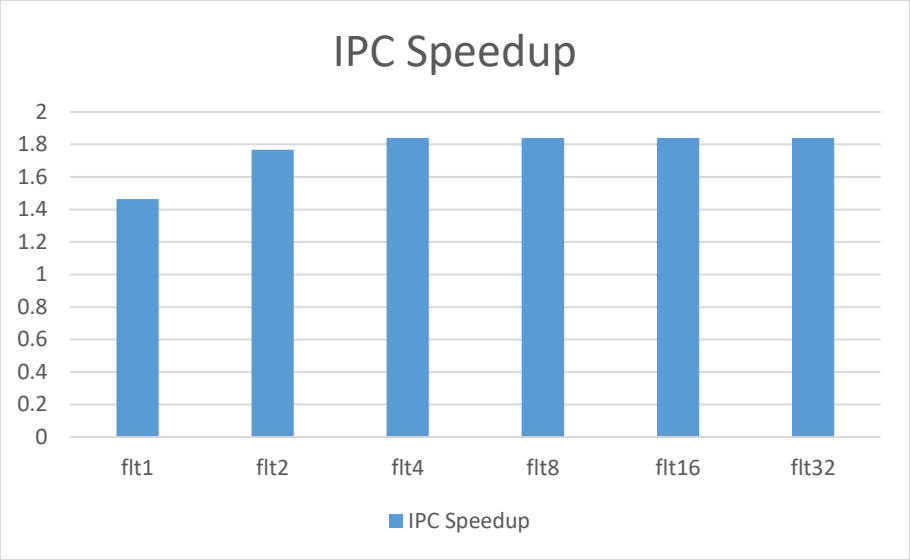


Figure 4.4: IPC Speedup for different stream buffer filter sizes with md32, mhsr8 and st1. The baseline used is Vldp16_baseline.

Following a similar process, the best configuration for the other three benchmarks is found and used for experimentation. Table 4.6 lists these configurations for each benchmark.

Table 4.6: Best stream buffer configurations for each benchmark

Benchmark	Max Depth	Additional MHSRs	Streams	Number of entries in each filter
Libquantum	32	8	1	16
LBM	32	16	4	32
Bwaves	16	8	4	16*
Leslie3d	64	8	4	16

* The best configuration for the Bwaves benchmark involves using 8 filters but with the consequent flow of the operation of the stream buffer, 8 filters causes fewer L2 MHSRs to be available for L1 demand misses, which should have higher priority. To prevent this, the next best configuration of 16 filters is used instead.

CHAPTER 5

Results

5.1 The Libquantum Custom Prefetcher

The PSM custom prefetcher for Libquantum provides great speedup (normalized IPC speedup between 3 - 4) and helps extract IPC potential that state-of-the-art prefetchers like IPCP (L1D prefetcher) and VLDP (L2/L3 prefetcher) by themselves are not able to, when running without PSM. Figure 5.1 shows that the PSM prefetcher performs very well and better than the other L1D prefetchers. In fact, this is true of all PSM + L1D prefetcher or PSM + stream buffer configurations for which the results are presented here – the PSM prefetcher as the only L1D prefetcher performs nearly as well by itself, as when paired up with another L1D prefetcher or the stream buffer, suggesting perhaps that some or all of the core's prefetchers could be turned off to save power. Figures 5.2, 5.3, and 5.4, show the prefetcher's performance for different clkC_wW, delayD, and queueQ configurations, respectively. These sweeps test the prefetcher's sensitivity to bandwidth, pipelined execution delay of the PSM-RF prefetcher design, and the sizes of the observation and intervention queues. It is evident from the plots that the prefetcher has good resistance to these changes and provides a speedup of more than 3 across all configurations.

Other than the accurate prefetches issued by the prefetcher, handling the timeliness requirement by setting up a good prefetch distance adaptively contributes significantly to the performance gains. Consider Figure 5.5, which shows the performance of different preset prefetch distance configurations versus the adaptive prefetching configuration. The plot shows that a prefetch distance of 40 or more ahead of the demand stream gives good performance speedup. The adaptive prefetching mechanism employed by Libquantum provides nearly the same speedup;

initially the prefetch distance is set to 40 and then different distances are adaptively finalized every time the region of interest is encountered (between 70-90 for region 1 and 60-90 for region 2 for the adaptive configuration in Figure 5.5).

Finally, Figures 5.6 and 5.7 show the reduction in misses-per-kilo-instructions (MPKI) and average load execution latency, respectively. The MPKI plot in Figure 5.6 shows a decrease in the number of load misses for one target load, but not much with the other one. However, a significant decrease in the average load latency leads to a boost in performance in both cases. Note that for certain configurations of the prefetchers, for the different benchmarks, the plots may show an average load latency of zero or close to zero in the graph. There are two reasons for this. First, for every dynamic load instruction, the L1 hit latency of 1 cycle was not included in its latency (just its miss cycles, if any), so a hit contibutes 0 cycles to the average load latency. Second, the average load latency was rounded from a real number to an integer, so that an average between 0 and 1 may round down to 0.

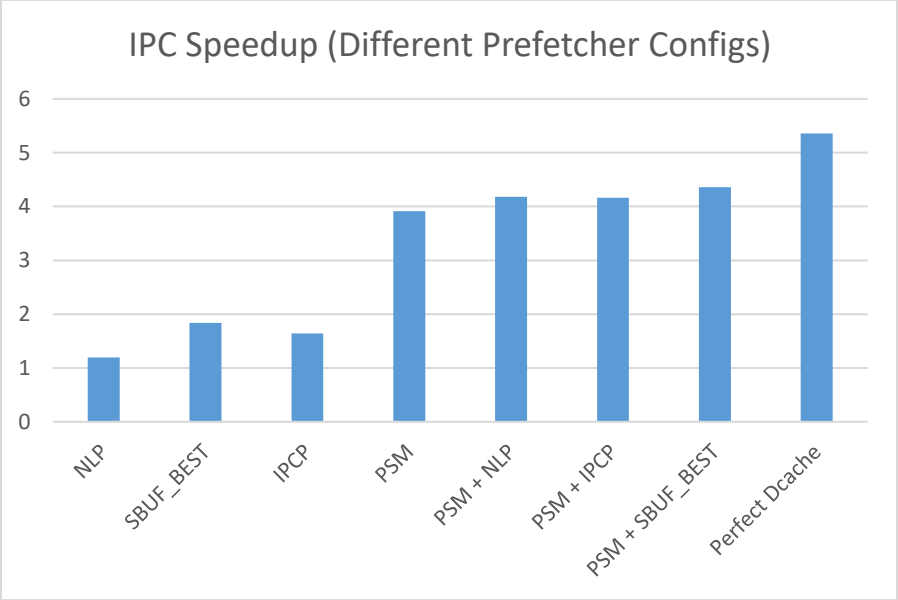


Figure 5.1: IPC speedups for Libquantum for different prefetcher combinations. The PSM configs all use clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

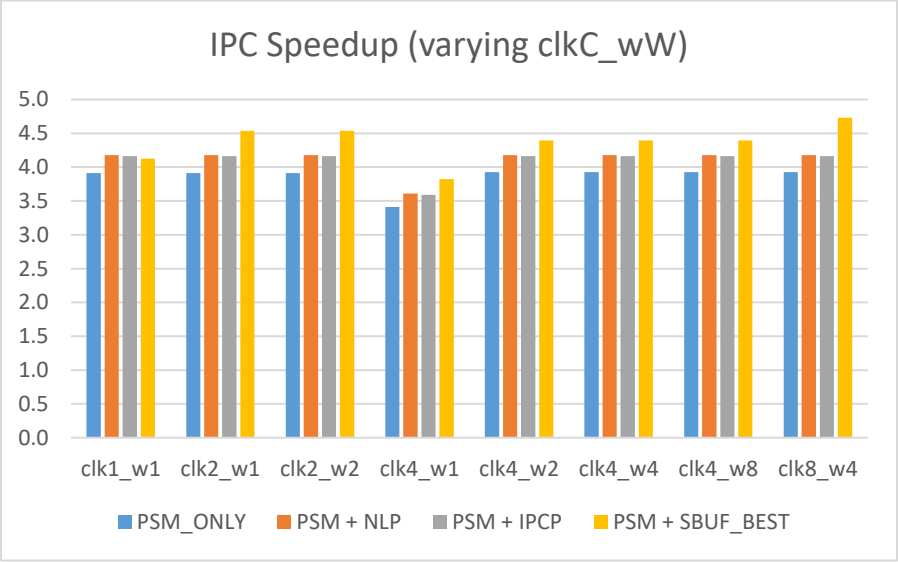


Figure 5.2: IPC speedups for Libquantum for different PSM clkC_wW configs with delay0 and queue32. The baseline used is Vldp16_baseline.

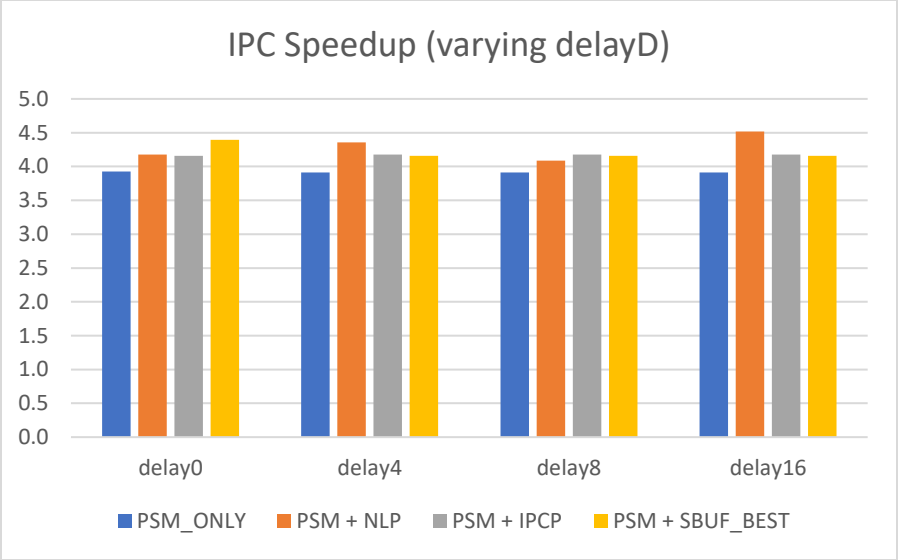


Figure 5.3: IPC speedups for Libquantum for different PSM delayD configs with clk4_w4 and queue32. The baseline used is Vldp16_baseline.

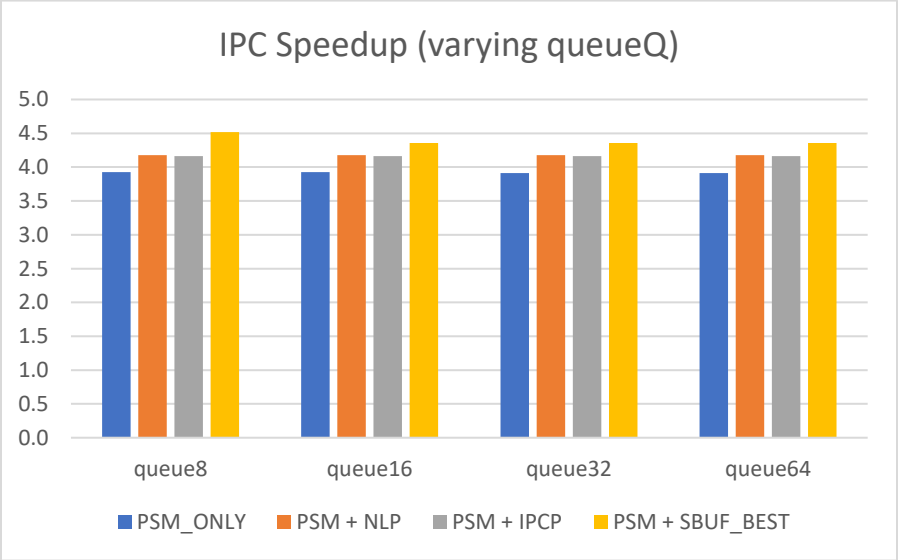


Figure 5.4: IPC speedups for Libquantum for different PSM queueQ configs with clk4_w4 and delay4. The baseline used is Vldp16_baseline.

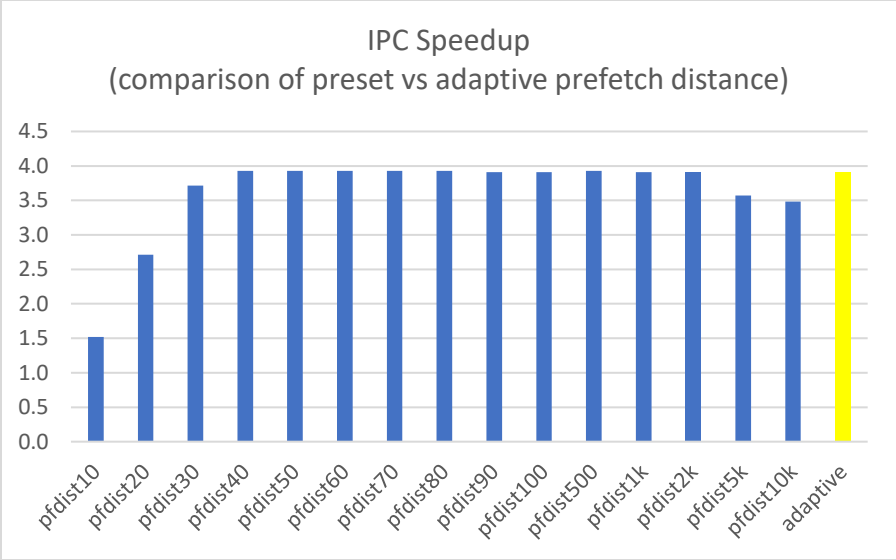


Figure 5.5: Comparison between preset and adaptive prefetch distance configurations` for Libquantum. All configurations use PSM + VLDP prefetchers, clk4_w4, delay4, and queue32. Adaptive final distances reached are between 60-90 across the two ROIs for the total of 6 times that they are encountered. The baseline used is Vldp16_baseline.

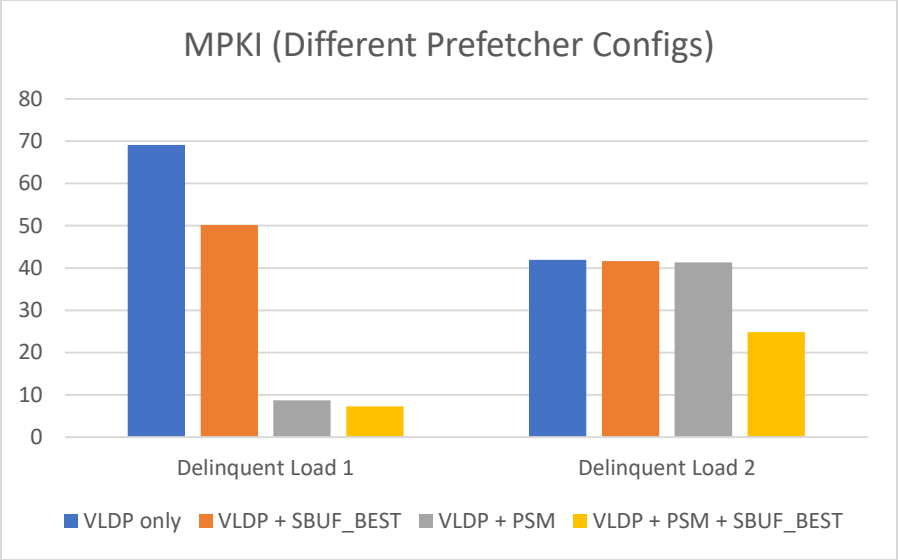


Figure 5.6: Trend for the number of load misses (in MPKI form) for the targeted loads for Libquantum across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

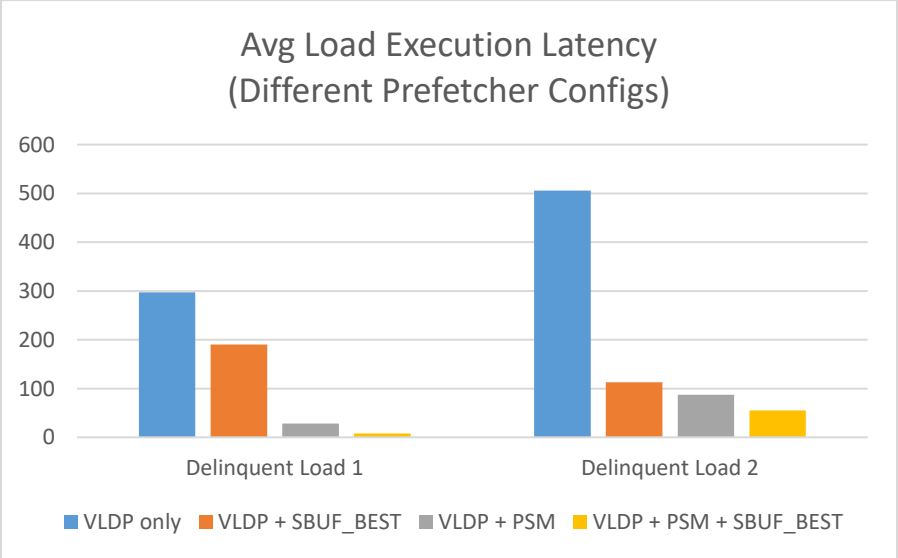


Figure 5.7: Trend for the average load execution latency for the target loads for Libquantum for different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

5.2 The LBM custom prefetcher

The custom prefetcher design for the LBM benchmark performs very well (IPC speedup of more than 4 across all prefetcher runs except with stream buffer, which is nearly 4) as is evident from Figure 5.8. It offers performance close to that with a perfect data cache (with or without a core prefetcher), as is evident from this plot, for the PSM + L1D prefetcher or PSM + stream buffer configurations except with stream buffer.

A process like the one for Libquantum was used to identify the best configuration for the stream buffer. Starting with an assumption of 16 filter entries, a sweep was performed with varying mdD, mhsrM, and stS. The configuration {md32, mhsr16, st4} was found to be the best for filt16. However, {md32, mhsr16, st4, filt32} showed more performance than {md32, mhsr16, st4, filt16} and hence, to correct the initial assumption, the final configuration was set to {md32, mhsr16, st4, filt32} which is shown as 'SBUF_BEST' in the plots that follow. However, when run with PSM prefetcher, this PSM + SBUF combination performs worse than PSM alone which is possibly because of the conflict between the stream buffer and PSM prefetcher with respect to resources such as cache and MHSRs.

All other PSM + L1D prefetchers or PSM + stream buffer configurations are bandwidth, execution delay, and queue size resistant, as shown in Figures 5.9, 5.11 and 5.12, respectively, and adaptive prefetching helps with that as shown in Figure 5.10. Experiments with changing the queue widths, delays, and sizes, show uniform speedup of about 4.5 (except for the PSM + stream buffer configuration). From Figures 5.13 and 5.14, which show the reduction in misses and load latencies (respectively) for different prefetcher configurations, we can see the effect that MLP-aware prefetching produces – all the latencies are evenly reduced when the PSM prefetcher is turned on. The VLDP-only configuration in Figure 5.14 shows the uneven latency reduction due to non-

awareness of the entire cluster of delinquent loads that need to be prefetched together; otherwise, the bottleneck shifts among these loads as is the case for the VLDP-only configuration.

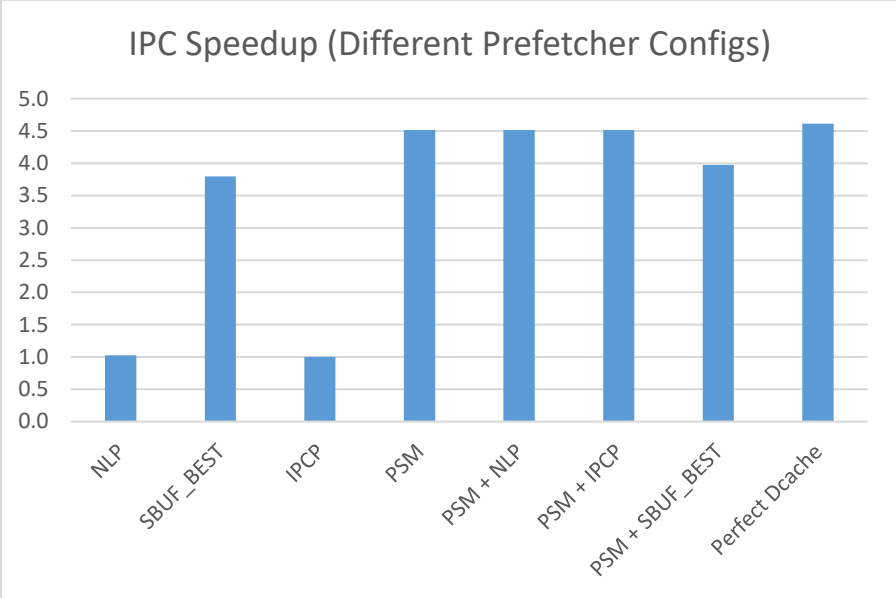


Figure 5.8: IPC speedups for LBM for different prefetcher combinations. The PSM configs all use clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

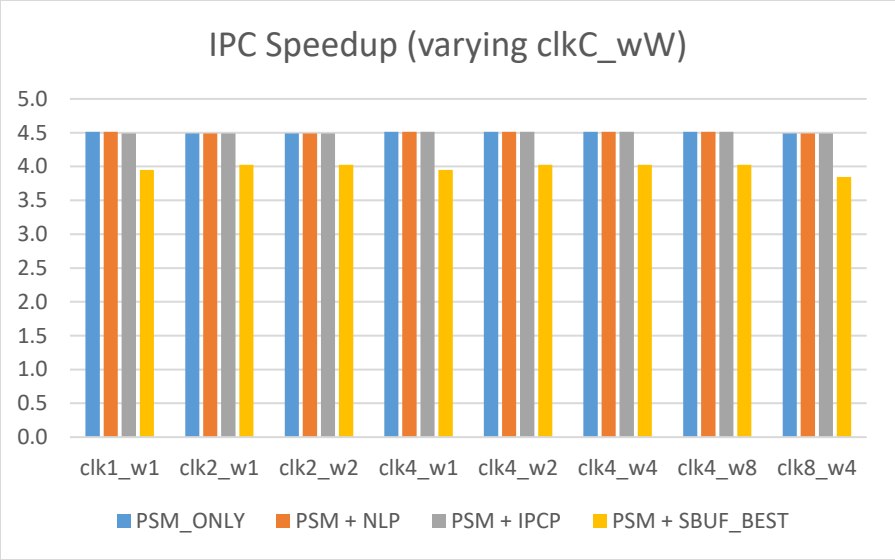


Figure 5.9: IPC speedups for LBM for different PSM clkC_wW configs with delay0 and queue32. The baseline used is Vldp16_baseline.

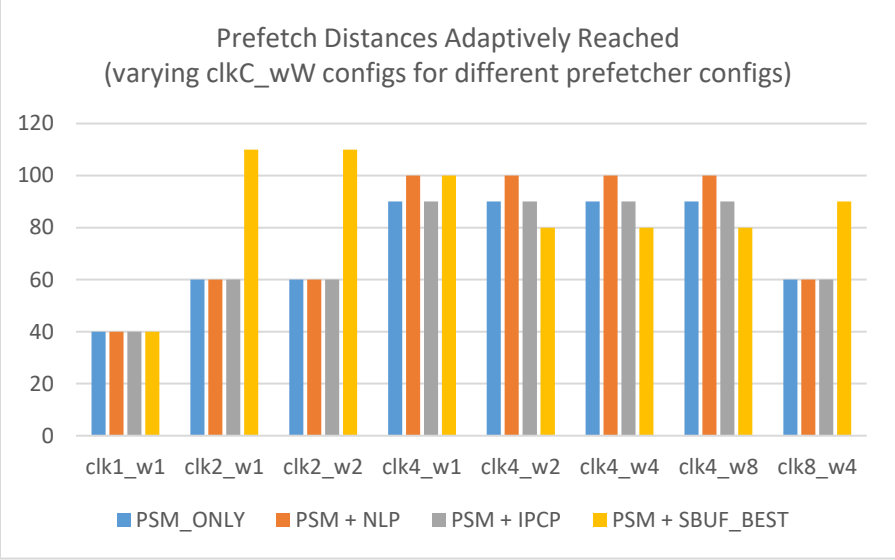


Figure 5.10: Prefetch distances adaptively reached for LBM for different PSM clkC_wW configs with delay0 and queue32. The baseline used is Vldp16_baseline.

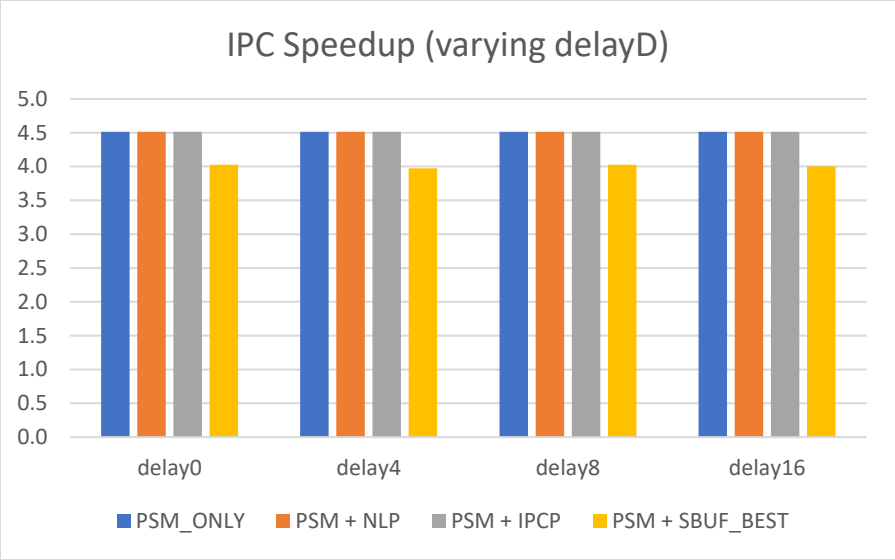


Figure 5.11: IPC speedups for LBM for different PSM delayD configs with clk4_w4 and queue32. The baseline used is Vldp16_baseline.

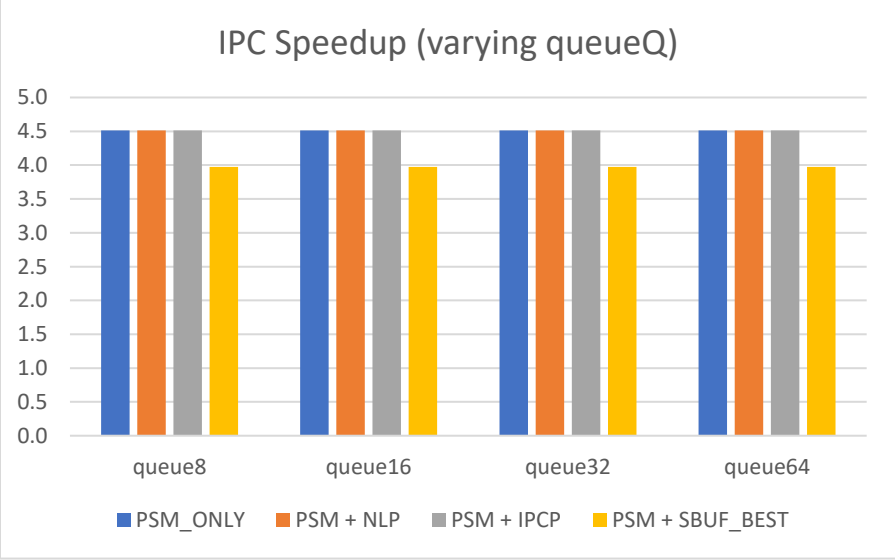


Figure 5.12: IPC speedups for LBM for different PSM queueQ configs with clk4_w4 and delay4. The baseline used is Vldp16_baseline.

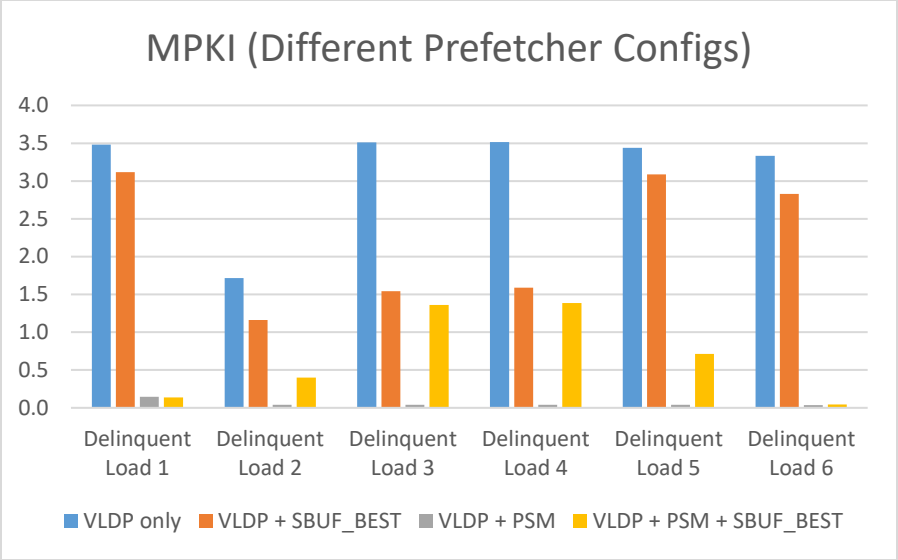


Figure 5.13: Trend for the number of load misses (in MPKI form) for the targeted loads for LBM across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

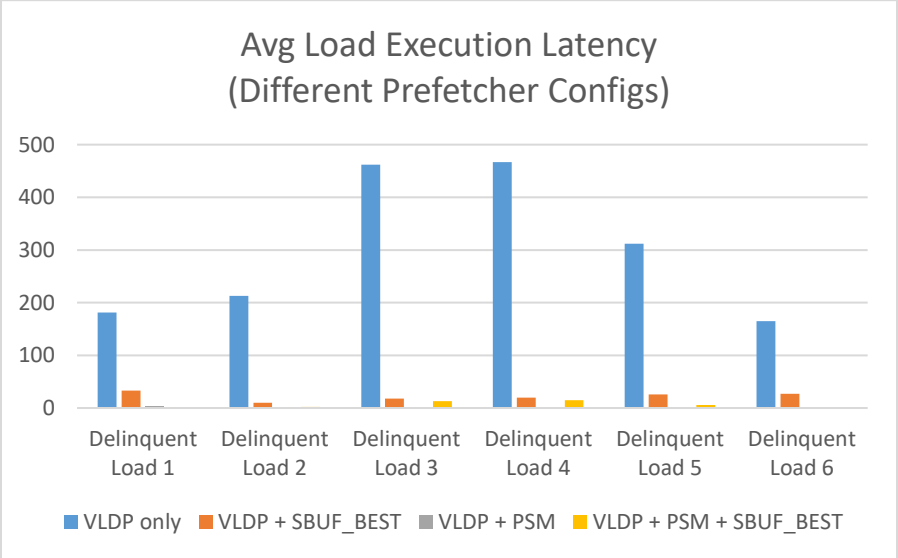


Figure 5.14: Trend for the average load execution latency for the targeted loads for LBM across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

5.3 The Bwaves custom prefetcher

The custom prefetcher design for Bwaves shows good speedup of between 1.5 – 2 for the different PSM configurations. While the PSM-only configuration does well by itself to provide good IPC performance for certain bandwidth, execution delay, and queue size configurations, having another L1D prefetcher or stream buffer with the PSM prefetcher helps boost the performance, as is evident from Figure 5.15. Figures 5.16, 5.17, and 5.18, show that the PSM prefetcher is resistant to varying bandwidth, delay, and queue size, with or without another L1D prefetcher or stream buffer active.

Plots in Figure 5.19 and Figure 5.20 show the reduction in misses and load latencies (respectively) for different prefetcher configurations. Reduction in the number of load misses and a reduction in the average load execution latency both lead to improved performance.

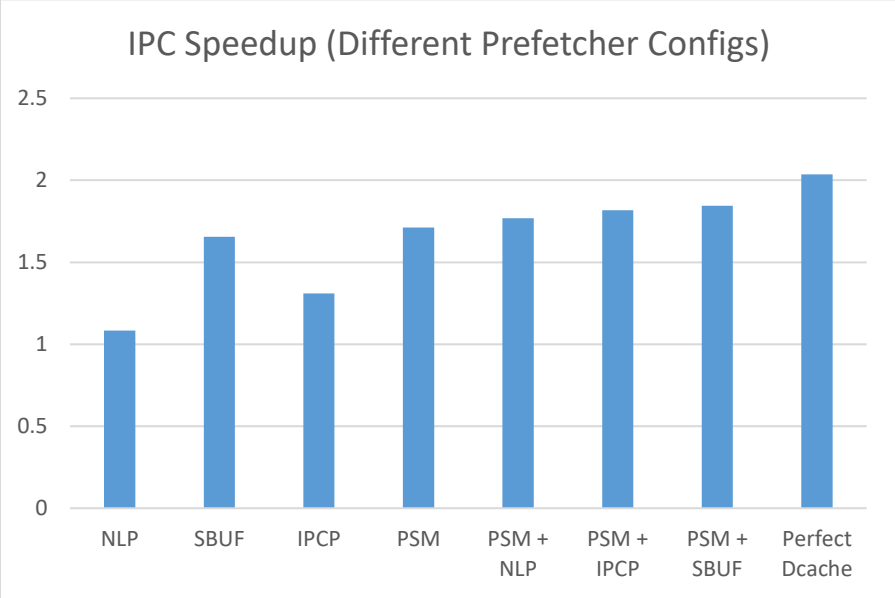


Figure 5.15: IPC speedups for Bwaves for different prefetcher combinations. The PSM configs all use clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

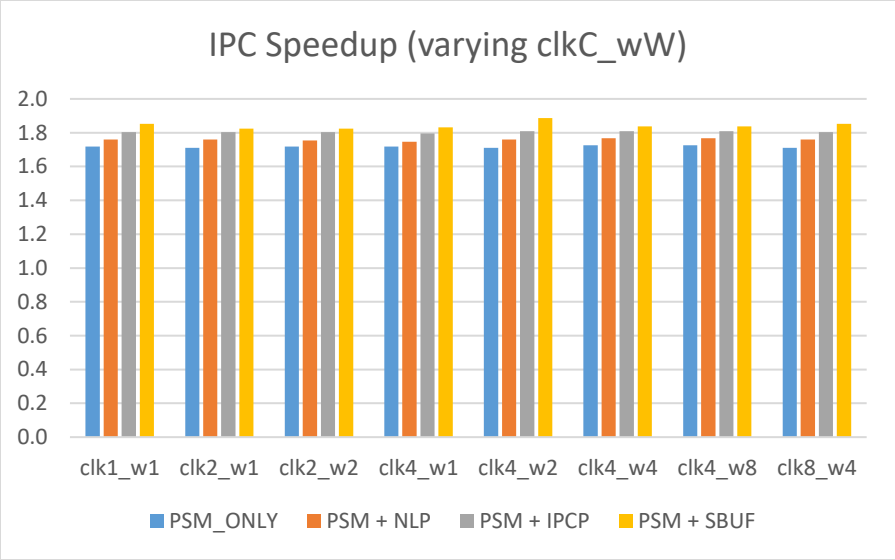


Figure 5.16: IPC speedups for Bwaves for different PSM clkC_wW configs with delay0 and queue32. The baseline used is Vldp16_baseline.

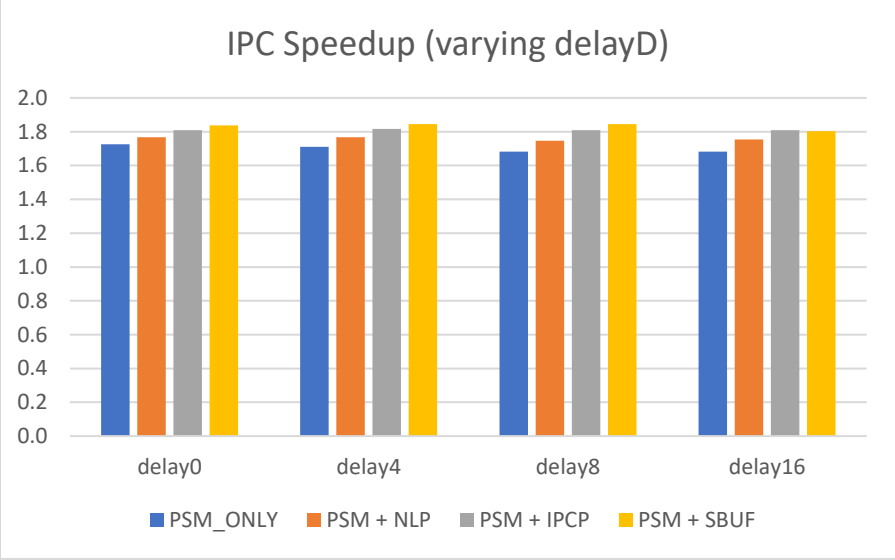


Figure 5.17: IPC speedups for Bwaves for different PSM delayD configs with clk4_w4 and queue32. The baseline used is Vldp16_baseline.

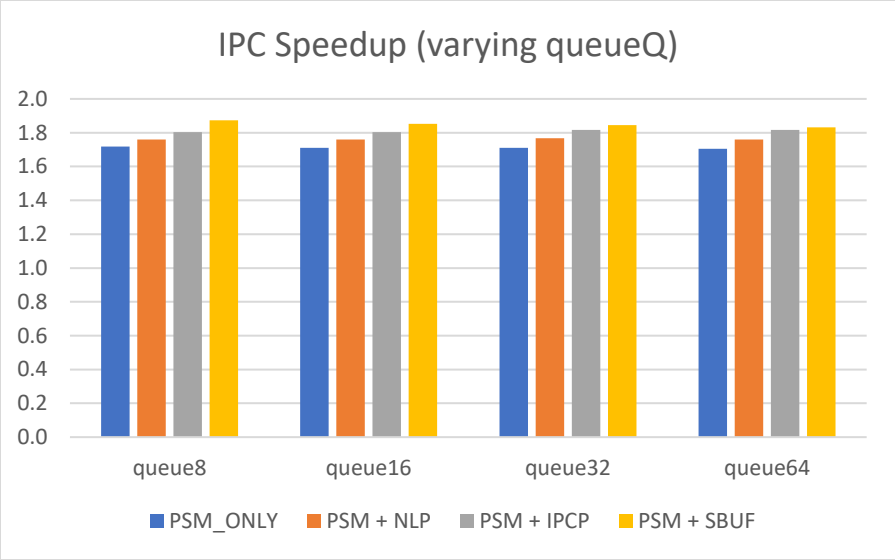


Figure 5.18: IPC speedups for Bwaves for different PSM queueQ configs with clk4_w4 and delay4. The baseline used is Vldp16_baseline.

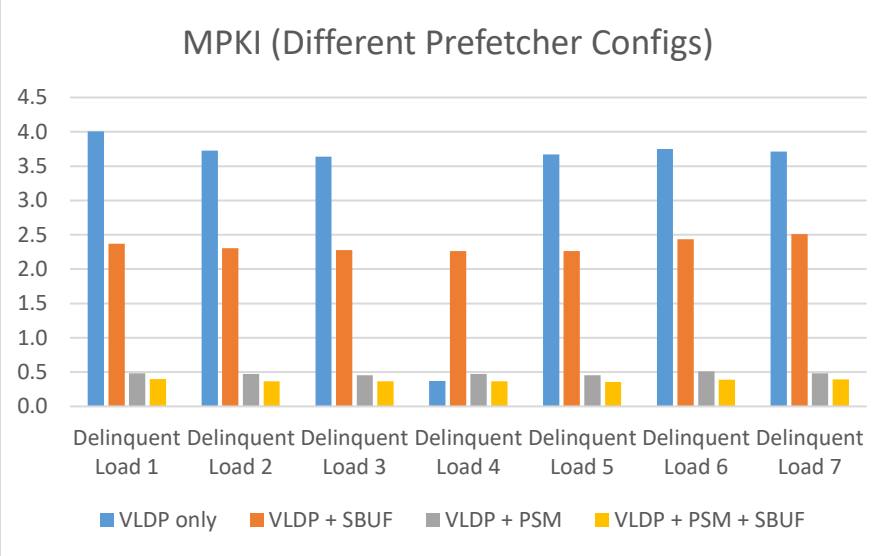


Figure 5.19: Trend for the number of load misses for the targeted loads for Bwaves across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

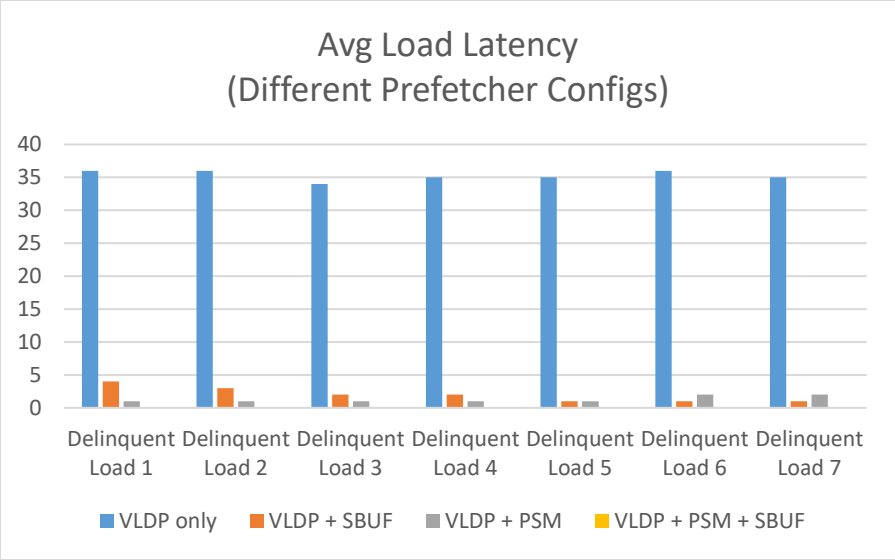


Figure 5.20: Trend for the average load execution latency for the targeted loads for Bwaves across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

5.4 The Leslie3d Custom Prefetcher

The Leslie3d custom prefetcher provides a speedup of 1.54 alone, and speedups between 1.86 to 2.24, when combined with L1D prefetchers and the stream buffer, as shown in Figure 5.21. Figures 5.22, 5.23, and 5.24, also demonstrate the prefetcher's resistance to varying bandwidth, delay and queue size. The performance boost can be attributed to a reduction in load misses and average load latencies as shown in Figure 5.25 and Figure 5.26, respectively. Also, the first three loads mentioned in these plots occur in the first ROI, the next two loads occur in the second ROI, while the last load occurs in the third ROI. For the first two ROIs, MLP-aware prefetching leads to an even latency reduction.

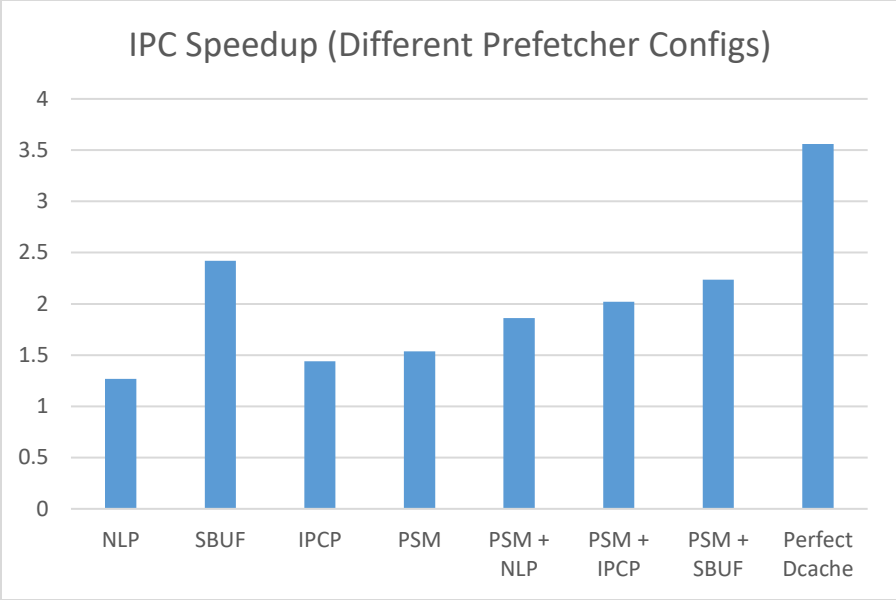


Figure 5.21: IPC speedups for Leslie3d for different prefetcher configurations. The PSM configs all use clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

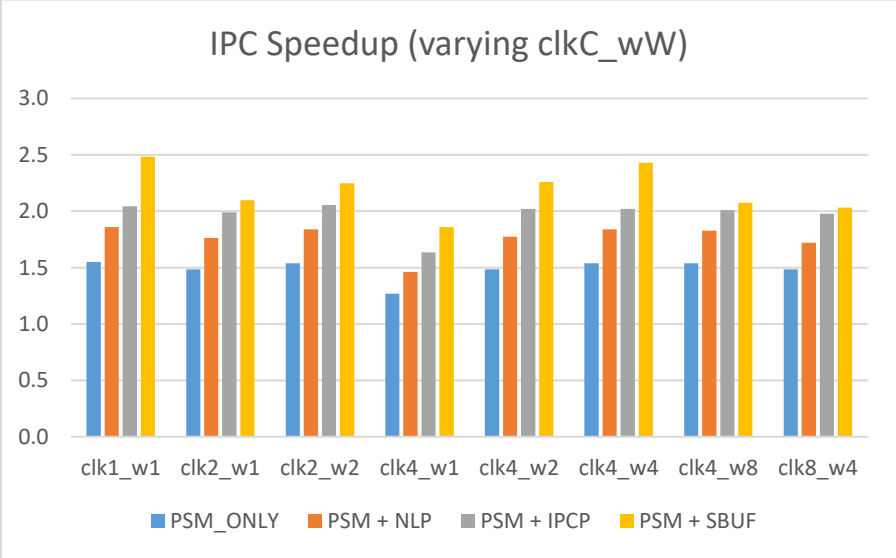


Figure 5.22: IPC speedups for Leslie3d for different PSM clkC_wW configs with delay0 and queue32. The baseline used is Vldp16_baseline.

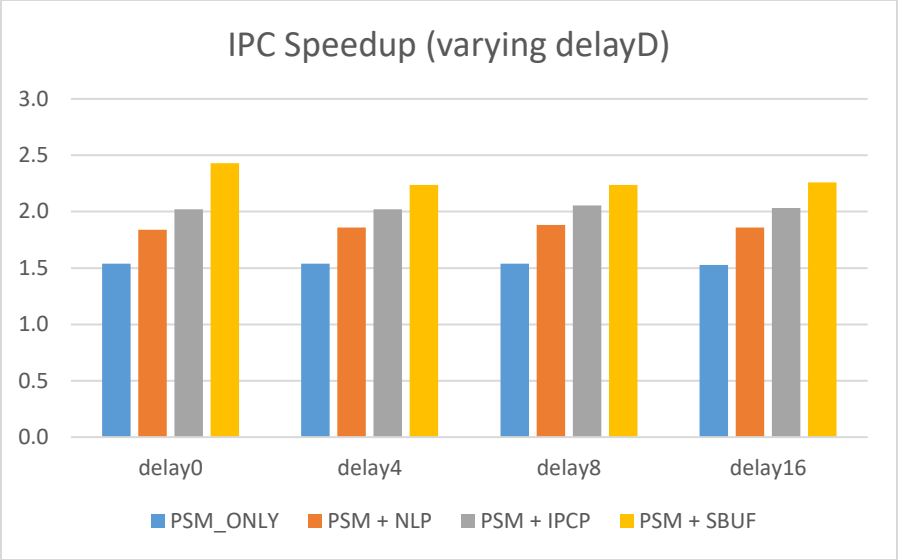


Figure 5.23: IPC speedups for Leslie3d for different PSM delayD configs with clk4_w4 and queue32. The baseline used is Vldp16_baseline.

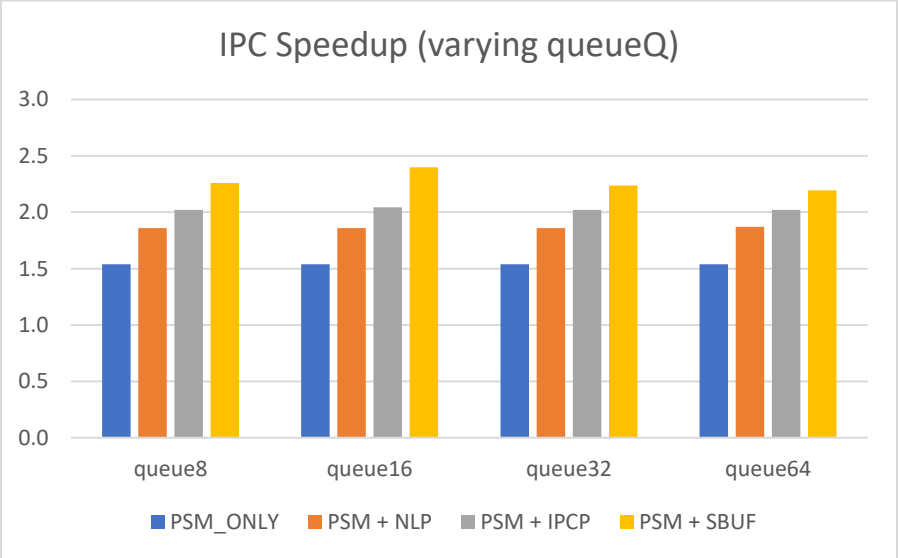


Figure 5.24: IPC speedups for Leslie3d for different PSM queueQ configs with clk4_w4 and delay4, over Vldp16_baseline. The baseline used is Vldp16_baseline.

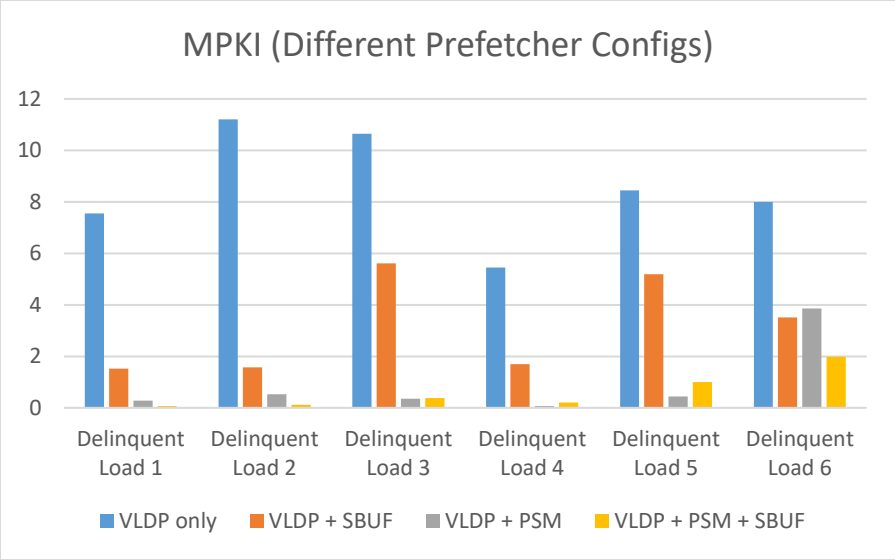


Figure 5.25: Trend for the number of load misses for the targeted loads for Leslie3d across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

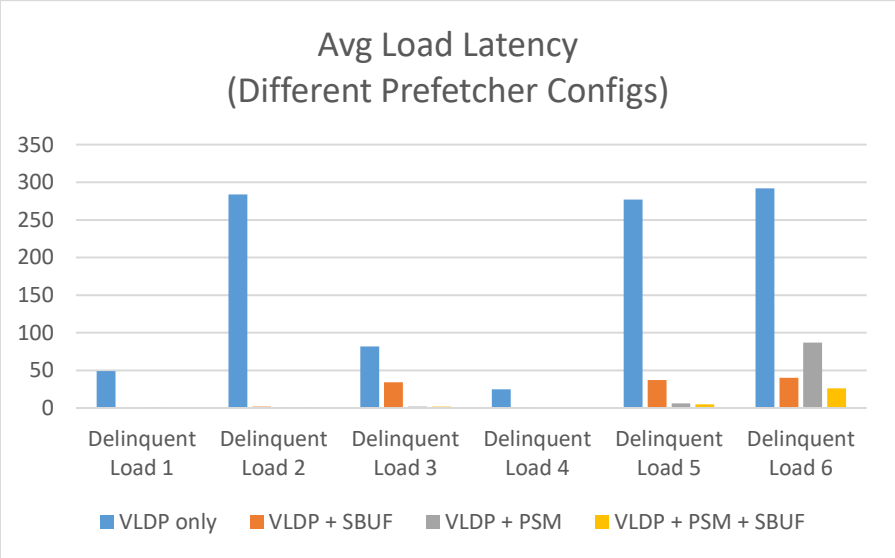


Figure 5.26: Trend for the average load execution latency for the targeted loads for Leslie3d across different prefetcher configurations. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is Vldp16_baseline.

5.5 Discussion

5.5.1 Effect of a benchmark's best stream buffer configuration on other benchmarks

So far, we have performed a series of sweeps to find the best stream buffer configuration for a certain benchmark by varying its configurable parameters. Table 5.1 shows these configurations for the respective benchmarks.

Table 5.1: Best stream buffer configurations for each benchmark (included from Chapter 4)

Benchmark	Max Depth	MHSRs	Streams	Number of entries in each filter
Libquantum	32	8	1	16
LBM	32	16	4	32
Bwaves	16	8	4	16
Leslie3d	64	8	4	16

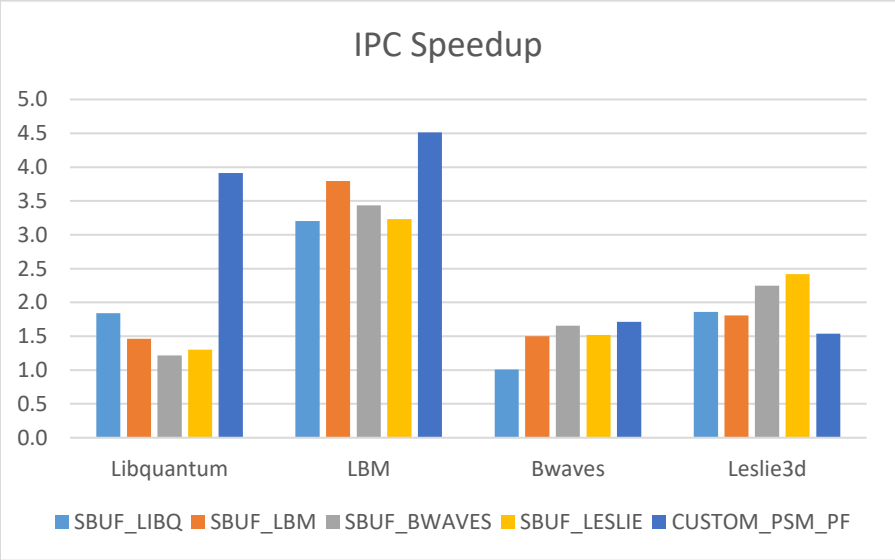


Figure 5.27: IPC speedup for different benchmarks with each benchmark's best stream buffer. The PSM configuration used is clk4_w4, delay4, and queue32.

Figure 5.27 presents a plot to demonstrate the effect of the best stream buffer configuration for one benchmark on other benchmarks. For example, for Libquantum, the configuration of stream buffer as mentioned in Table 5.1 provides maximum speedup for it but not for other benchmarks. For LBM and Bwaves, it is the least effective of all the other stream buffer configurations, while for Leslie3d it is the third choice out of four choices of stream buffer configurations. This shows that using a common stream buffer configuration across all benchmarks may not provide the best speedup and in choosing the best configuration for each benchmark, there is a kind of customization involved, to get the most performance.

The custom PSM prefetcher for each respective benchmark performs the best for Libquantum and LBM, a bit more than the top stream buffer configuration for Bwaves, and not so good with Leslie3d, likely due to non-ROI regions getting more benefit from the stream buffer than the PSM prefetcher. It is left to future work to confirm whether or not other loads in Leslie3d, outside of the targeted ROIs, are the reason PSM underperforms the stream buffers, and to determine if these other loads can also be targeted. Figures 5.25 and 5.26, in Section 5.4, show that on the whole PSM is effective in reducing misses and latency for the ROIs. On the other hand, Figure 5.25 shows that load 6 is still problematic for both the stream buffer (MPKI reduced from 8 to 3.5) and PSM (MPKI reduced from 8 to 3.9). Figure 5.26 shows the stream buffer is better at reducing load 6's average latency (from 292 down to 40 cycles) than PSM is (from 292 down to 87 cycles).

5.5.2 Effect of varying MHSR occupancy thresholds on the performance of core prefetchers

For sections 5.1 to 5.4, we constrained the core prefetchers to prefetch only if certain MHSR occupancy thresholds were met. These restrictions were presented in Table 4.5. As mentioned previously, we placed these restrictions in order to give higher priority to demand misses over prefetch misses. We now investigate the effect of throttling these thresholds on the performance of the prefetchers, as a case study. We vary the thresholds for VLDP, NLP, and IPCP prefetchers for the Leslie3d benchmark to illustrate the effect of such variation on the IPC; this variation is shown in Figure 5.28. Table 5.2 describes the new configurations and serves as the key for this plot.

Also, the baseline of all the plots herein is changed to a 'no-prefetcher' 721sim version (No_pf_baseline) which means that no prefetcher for any level of cache, not even VLDP, is active when this baseline is run for any benchmark. This change in baseline is necessary because Vldp16_baseline, which was being used for IPC speedup comparisons in sections 5.1 to 5.4, consistently used an MHSR occupancy threshold of 16, i.e., a configuration of vldp16. For experiments in this section, however, this threshold is changed to show its effect on IPC.

Table 5.2 Key for new MHSR occupancy thresholds on the core prefetchers

Prefetcher	**Previous Configuration***	**Current Configuration (Throttled)**	**Prefetches Into**	**Prefetch Only If**
VLDP	vldp16	vldpN	L2 cache	**Number of L2 MHSRs occupied < N** Number of L3 MHSRs occupied < 96[+]
			L3 cache	Number of L3 MHSRs occupied < 96
NLP	nlp16	nlpN	L1 D-cache	**Number of L1 D-cache MHSRs occupied < N**
IPCP	ipcp16	ipcpN	L1 D-cache	**Number of L1 D-cache MHSRs occupied < N**

* Previously used in sections 5.1 to 5.4.

[+] This limit on L3 MHSR occupancy is non-configurable and fixed for VLDP as well as for any stream buffer configuration used throughout this thesis. Only the L2 MHSR occupancy threshold is configurable and is changed.

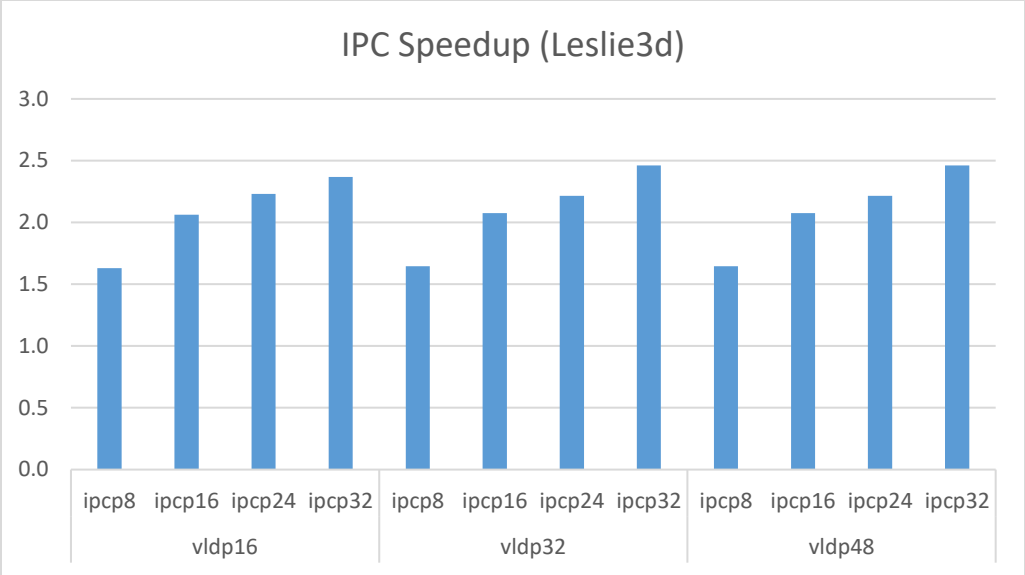


Figure 5.28: IPC Speedup for different configurations of IPCP for Leslie3d (varying MHSR occupancy thresholds on IPCP and VLDP). The baseline used is No_pf_baseline.

From Figure 5.28, we see that reducing the MHSR occupancy thresholds makes the prefetchers perform better. The best speedup is 2.46, obtained for two configurations – {vldp32, ipcp32} and {vldp48, ipcp32}. Similarly, changing the MHSR occupancy threshold for the prefetchers across other benchmarks results in different speedups. In most cases, the best configuration for the IPCP and NLP prefetchers is obtained when the MSHR occupancy restriction is completely relaxed, i.e., the prefetcher issues a prefetch as long as there is at least one free MHSR (although IPCP is still opportunistic in sharing the D-cache ports with the core pipeline). Note that we change VLDP's MHSR occupancy threshold only up to a maximum of 48, and never beyond that, to effectively reserve some MHSRs for L2 demand misses. For the relaxed stream buffer configurations, we use 'vldp16' as the MHSR threshold configuration for VLDP consistently and only change the MHSR occupancy threshold for the stream buffer.

The plots that follow, include the IPCP, NLP, and stream buffer configurations for which some of the MHSR occupancy thresholds are relaxed. These are respectively termed as IPCP_RELAXED, NLP_RELAXED and SBUF_RELAXED, and described in Tables 5.3 and 5.4. For purposes of comparing their performance with the PSM prefetcher configurations, plots from earlier sections have also been included and augmented with relaxed MHSR variants.

Table 5.3 Description of IPCP_RELAXED and NLP_RELAXED prefetcher configurations where the MHSR occupancy restrictions are relaxed, as shown for each respective benchmark

	NLP_RELAXED	IPCP_RELAXED
Libquantum	nlp32, vldp16	ipcp32, vldp16
LBM	nlp32, vldp16	ipcp32, vldp16
Bwaves	nlp32, vldp16	ipcp32, vldp16
Leslie3d	nlp32, vldp48	ipcp32, vldp48

Table 5.4 Description for stream buffer configuration 'SBUF_RELAXED'

Prefetcher configurations used	**Previous Config:** **Prefetch Only If**	**Current Config:** **Prefetch Only If**
SBUF_BEST^ + vldp16	Number of L2 MHSRs occupied < 48 Number of L3 MHSRs occupied < 96	**Number of L2 MHSRs occupied < 64*** Number of L3 MHSRs occupied < 96

^ Best {mdD, mhsrM, filtF, stS} configuration as determined in section 4.3 (Table 4.6)

* This is true for all benchmarks except for LBM, where this is set to 48 instead of 64. This is because completely relaxing the MHSR restriction and setting the upper limit to 64 causes contention with respect to the L2 MHSR usage, between the L2 prefetch misses (triggered by VLDP and stream buffer) and the L2 demand misses (triggered by L1 demand accesses).

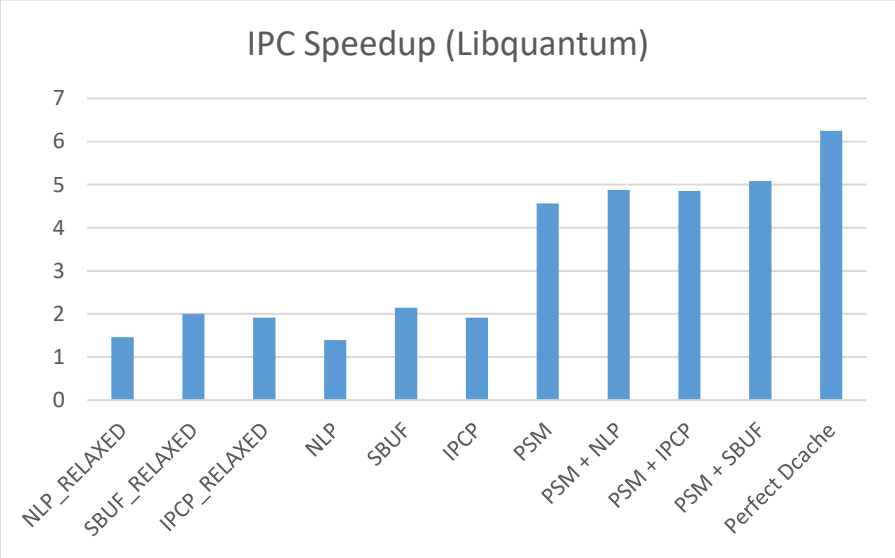


Figure 5.29: IPC Speedup for different prefetcher configurations for Libquantum benchmark. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is No_pf_baseline.

As shown in Figure 5.29, the Libquantum benchmark does not show any significant change for the IPCP_RELAXED, NLP_RELAXED, and SBUF_RELAXED configurations, as compared to their respective counterparts with restricted MHSR usage. The PSM configurations outperform the standalone L1D prefetchers and the stream buffer for the runs. The Libquantum ROIs are loops with a short loop body due to which the frequency of loads is high. The prefetching for such loops

needs to be very timely. With IPCP, the hybrid prefetcher's stream prefetcher gains control over the prefetching (as opposed to the constant stride and complex stride) when a stream is detected. So, most of its prefetches are issued by the stream prefetcher. Although it is a simple strided pattern with stride 1, the effective prefetch distance between the demand and prefetch streams achieved by this prefetcher isn't enough to ensure timeliness (prefetch degree of 6 as per the Championship prefetcher source code), even when the MHSR occupancy restriction is relaxed.

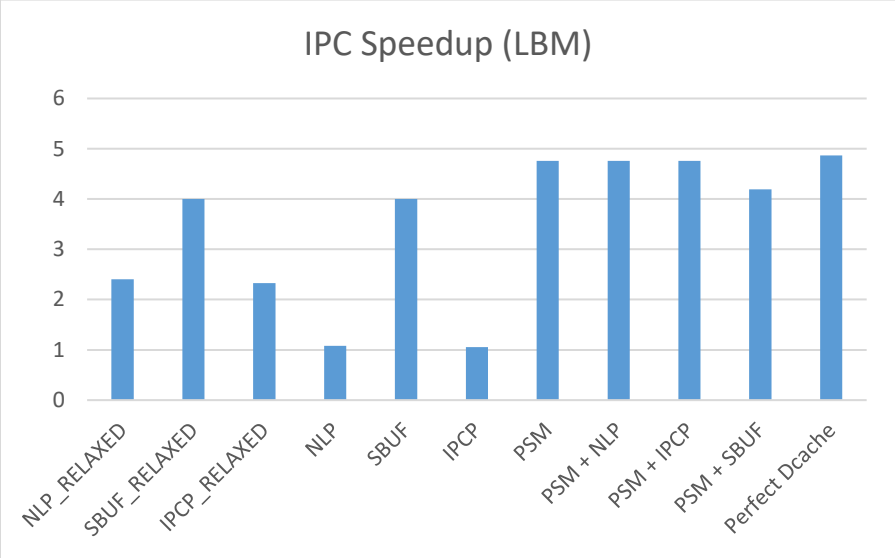


Figure 5.30: IPC Speedup for different prefetcher configurations for LBM benchmark. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is No_pf_baseline.

From Figure 5.30, the 'NLP_RELAXED' and 'IPCP_RELAXED' configurations do better than their restricted counterparts but are outperformed by the 'PSM' configuration.

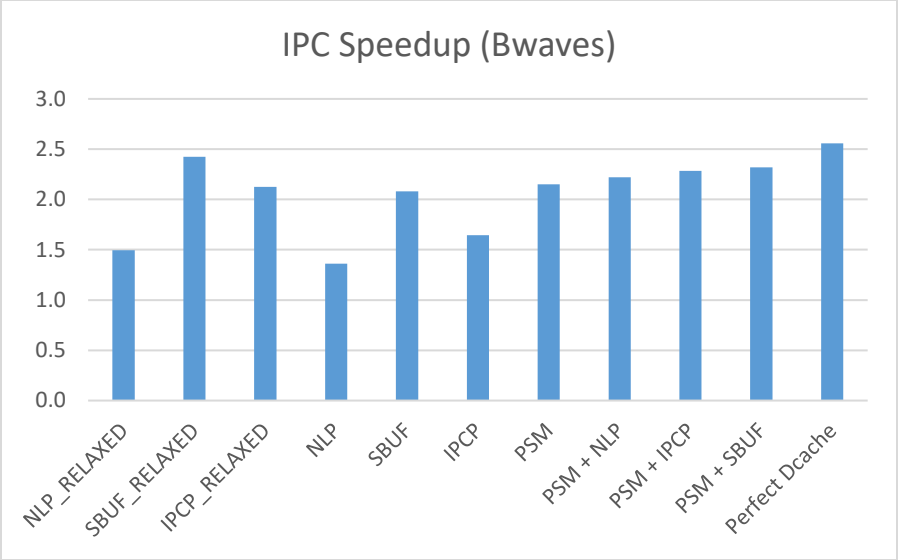


Figure 5.31: IPC Speedup for different prefetcher configurations for Bwaves benchmark. The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is No_pf_baseline.

As shown in the Figure 5.31, the ‘SBUF_RELAXED’ configuration for the Bwaves benchmark outperforms every other prefetcher configuration. The ‘IPCP_RELAXED’ configuration shows significant improvement with MHSR restrictions relaxed. The ‘PSM’ configuration provides speedup less than that of the ‘SBUF_RELAXED’ configuration but when combined with another L1D prefetcher, the performance gap gets shorter.

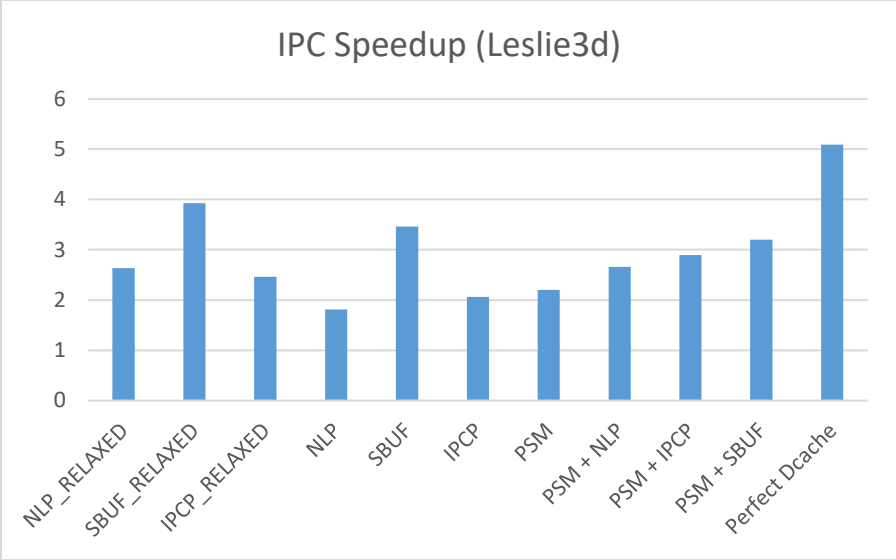


Figure 5.32: IPC Speedup for different prefetcher configurations for Leslie3d benchmark. The PSM configuration uses clk4_w4, delay4, and queue32 configuration. The baseline used is No_pf_baseline.

From the plot in Figure 5.32, it is evident that similar to Bwaves, Leslie3d gets the most benefit from the relaxed stream buffer configuration which outperforms all other prefetcher configurations. 'IPCP_RELAXED' shows more speedup than its counterpart configuration with MHSR restriction and also more than PSM. One reason for this could be that IPCP is active throughout the duration of the simpoint program whereas the custom prefetcher targets the ROIs only. While the delinquent loads in the ROIs are a major reason for the low baseline IPC, improving the latencies of the non-ROI regions, too, seems to benefit Leslie3d to a good extent. This could also be why the NLP_RELAXED outperforms PSM. When combined, the prefetchers' interaction with PSM seems to enhance the overall performance so that the speedup that results is greater than either of the prefetchers. This is especially noticeable with the PSM + IPCP configuration.

5.5.3 Effect of disabling L2/L3 VLDP prefetcher on the performance of L1D prefetchers and stream buffer

The VLDP prefetcher has served as the L2/L3 prefetcher for every configuration presented so far. In order to get an idea about the difference that the VLDP prefetcher makes in the various prefetcher configurations, we disable the VLDP prefetcher from the configurations used in sections 5.1 to 5.4, keeping everything else the same. Figure 5.33 shows speedups for the various prefetcher runs, with and without VLDP.

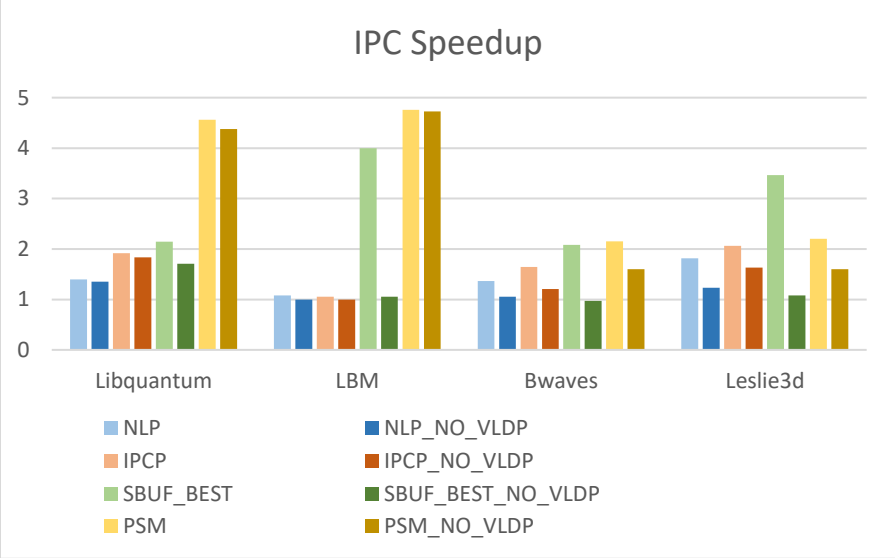


Figure 5.33: IPC Speedup for different L1D prefetchers and stream buffers when run standalone across all benchmarks (No L2/L3 prefetcher). The PSM configuration uses clk4_w4, delay4, and queue32. The baseline used is No_pf_baseline.

From Figure 5.33, disabling the VLDP prefetcher reduces the IPC across the board. For the Libquantum and LBM prefetcher runs without VLDP, the PSM prefetcher shows good resistance to the absence of VLDP and outperforms the core prefetchers. For the Bwaves prefetcher

runs without VLDP, while the IPC drops for the PSM prefetcher in the absence of VLDP, it still performs better than other prefetchers. For the Leslie3d prefetcher runs without VLDP, the PSM prefetchers still perform better than NLP and stream buffer, and comparable to IPCP. Since PSM prefetches are surgical, it suggests that VLDP is important to increase the timeliness of PSM prefetches. The stream buffer shows significant drop in performance when VLDP prefetcher is disabled, suggesting a strong dependence on VLDP (or an L2/L3 prefetcher, in general), which needs to be investigated in future work.

CHAPTER 6

Conclusion and Future work

6.1 Conclusion

One of the major issues faced by conventional CPU cores is low performance due to long latency load misses, especially at the lowest levels in the memory hierarchy. One way to solve the problem of high latency load misses is to use prefetching to request the data that is likely to be used in the future. Good prefetching, in terms of accuracy, coverage, and timeliness, leads to higher load latency reduction and high gains in performance from the core. State-of-the-art prefetchers like IPCP and VLDP work well to reduce these latencies but suffer from problems that cause low performance gains. One of the obstacles is that they are not aware of the existing memory-level parallelism in the cache access patterns leading to uneven prefetching and uneven latency reduction across a cluster of loads. Also, they don't seem to adapt well to the complex strided pattern of the accesses.

We propose the development of highly customized benchmark-specific prefetchers that are tailored to the individual workloads that run on the core. Profiling the workload and the assembly of its source code reveals regions of interest where prefetching can work to reduce high cache miss latencies (or even the number of cache misses occurring, in some cases). Once the region of interest is known, the address computation algorithm followed in those regions can be mimicked in hardware so that highly accurate prefetches can be generated to effectively prefetch the demand access stream. If complex nested loops exist in the region, we mimic exactly the loop structures including the number of iterations of the different levels of loop and updates to variables that are involved in the address computation. Such a highly custom prefetcher requires support from the

core, such as information regarding the number of iterations, indications to start and stop prefetching for these regions, and other miscellaneous information that helps stay ahead of the demand access stream for producing timely prefetches. ‘Post-Silicon Microarchitecture’ is a paradigm that facilitates such interfacing and is the focus of all experiments performed in simulation to show its viability and effectiveness.

Post-Silicon Microarchitecture (PSM) is a paradigm that provides a way to synthesize and interface application-specific microarchitectural interventions after fabrication. The core and reconfigurable fabric coupled via a flexible interface provides a means to boost performance of individual workloads that run on the core but aren’t accelerated for IPC as much by the dedicated general-purpose predictors and prefetchers. Since components can be synthesized as and when needed, it increases the appeal for the development of highly custom optimizations that could not be justified on the core as dedicated components due to their narrow applicability. Consequently, custom prefetchers that mimic the regions with delinquent loads (ROIs) in the workloads can be synthesized to provide high IPC gains.

To ensure timely prefetching, we use an adaptive mechanism that sets up an appropriate prefetch distance between the demand and prefetch streams such that the prefetcher can adapt to different PSM configurations for clock frequencies, interface queue sizes, and execution latencies of the PSM-RF design. Further, the prefetcher is made aware of the memory level parallelism so that it prefetches for the entire cluster of cache missing loads. This can also mean that if, for certain workloads, the intervention queue frequently lacks space to accommodate additional prefetches, a prefetch dropping policy can be employed such as in the case of Bwaves and Leslie3d benchmarks, while ensuring that the prefetches are dropped as entire clusters rather than individual load prefetch

drops. This leads to an even load latency reduction across the entire cluster of loads and helps speed up the retirement of the loop iterations.

Overall, this leads to high gains in performance for the individual benchmarks. The Libquantum benchmark shows a normalized IPC speedup of 3 – 4 (over a baseline with the VLDP L2/L3 prefetcher, with speedup 1) for different configurations of the PSM prefetcher by itself and also when working alongside a simple next-2-line prefetcher, a state-of-the-art IPCP prefetcher, or a stream buffer. The LBM benchmark shows a speedup of 4 or more across all PSM configurations, with minimal differences across combinations of the PSM prefetcher with different core L1D prefetchers and stream buffer. The Bwaves benchmark shows speedup of 1.5 – 2 regardless of whether it is running standalone or with another L1D prefetcher or stream buffer. Finally, the Leslie3d benchmark shows a speedup of about 1.5 standalone and between 1.8 to 2.3 when running alongside core L1D prefetchers or stream buffer. The results also point to the fact that for certain workloads and PSM configurations, the PSM prefetcher by itself performs well enough to remove the necessity of having additional core prefetchers. It is often seen that for many PSM configurations, a combination of the PSM prefetcher with a core L1D prefetcher of the stream buffer, performs better than either of the two standalones.

We perform experiments to dive deeper into the roles of certain aspects of the prefetchers, specifically, the role of MHSR occupancy limits and the presence of the VLDP L2/L3 prefetcher. To provide higher priority to demand misses over prefetch misses, we apply certain MHSR occupancy restrictions on the core prefetchers when first discussing the performance of PSM prefetchers (Sections 5.1 to 5.4). However, we also explore the impact of varying the MHSR occupancy thresholds and observe that the performance of the core prefetchers increases as we reduce the restrictions. The best configuration for the IPCP and NLP prefetchers is obtained when

the L1 D-cache MHSR occupancy restriction is completely relaxed, i.e., the prefetcher issues a prefetch as long as there is at least one free L1 D-cache MHSR. The stream buffer shows an increase in performance for the Bwaves and Leslie3d benchmarks when some of these restrictions are relaxed. However, across a broader spectrum of workloads, it may not be a good policy to let the prefetchers run freely, without assigning higher priority to demand misses in some way or another. Results from disabling the VLDP prefetcher suggest that it plays a role in increasing the timeliness of the PSM prefetches, although the PSM prefetcher by itself shows good resistance to the absence of VLDP.

From experiments with the stream buffer, we see that it offers great performance boosts for certain benchmarks, especially for Bwaves and Leslie3d (in the presence of VLDP prefetcher for L2/L3), albeit when its configuration is customized for each benchmark. However, stream buffers require additional storage which can be very high for multiple streams and greater depths of 32 or 64. Lower depths may not provide good prefetch distance, however, and lead to timeliness issues with the prefetches. Further, the stream buffer design that works well for one benchmark may not be ideal for other benchmarks. As a result, to get the most performance out of individual workloads, a custom stream buffer configuration is needed which further strengthens the proposal for custom microarchitecture development.

Through all the results and insights, the development of custom prefetchers tuned to specific workloads and the high gains in performance that they produce in various settings show the viability and appeal of using a Post-Silicon Microarchitecture.

6.2 Future Work

The prefetcher interface for the baseline 721sim simulator, that was created to facilitate the interfacing of arbitrary L1 d-cache prefetchers (Section 3.1), can be extended to provide notifications for when a certain cache fill occurs. This can be useful to some prefetchers that use this information to update key structures during their operation. Further, a similar interface can be modeled for L2/L3 prefetchers as well to facilitate the use of different prefetchers at all levels of the cache hierarchy.

The custom prefetchers show great promise for mitigating the latencies of the delinquent loads. There can be improvements made to this design, which can also benefit arbitrary prefetchers that are modeled and interfaced with the core simulator. Inclusion of a 'prefetch bit' in each cache line can help find which prefetched lines are useful and ones which get evicted without being requested by the demand stream. This kind of metadata can help get more information about metrics such as accuracy, coverage, and timeliness. Further, this can also inform the adaptive mechanism and make it more aggressive by giving it more direct feedback about the prefetcher's performance rather than indirectly through the retirement of instruction PCs. This can be taken a step further to make the prefetcher more sophisticated whereby it keeps track of this data and dynamically changes its aggressiveness in terms of prefetch degree and distance. Such sophistication can help the PSM prefetcher (and any arbitrary core prefetcher as well) for programs that go through different phases within the region of interest and improve the adaptive prefetching mechanism to yield good prefetch distances.

Also, more use cases that demonstrate different classes of demand access patterns could be explored apart from the computational prefetchers developed in this thesis. For example, for demand accesses that are spatially correlated and show repeating patterns in the relative offsets of

demanded addresses, we could design a custom prefetcher that learns these correlations specifically for the targeted workload's region of interest and helps reduce the average delinquent load latency for the region. A comparison of such a prefetcher could be made with the SMS prefetcher [15] that works well with the spatially correlated demand access streams. In fact, a design similar to the stream buffer can be a good candidate for PSM exploration since a custom stream buffer design performs well with each benchmark. Such studies could help reveal more benefits of using the PSM interface.

The results presented in thesis can inspire further lines of inquiry about the stream buffer, especially to understand the reason behind the strong dependence of stream buffers on the VLDP L2/L3 prefetcher. The reason for the Leslie3d PSM prefetcher's low IPC gains for certain configurations should also be investigated.

REFERENCES

[1] S. Pakalapati and B. Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching," 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 2020, pp. 118-131, doi: 10.1109/ISCA45697.2020.00021.

[2] S. Pakalapati and B. Panda, “Bouquet of Instruction Pointers: Instruction Pointer Classifier based Hardware Prefetching.”, 3rd Data Prefetching Championship, 2019.

[3] O. Mutlu, J. Stark, C. Wilkerson and Y. N. Patt, "Runahead execution: An effective alternative to large instruction windows," in IEEE Micro, vol. 23, no. 6, pp. 20-25, Nov.-Dec. 2003, doi: 10.1109/MM.2003.1261383.

[4] Z. Purser, K. Sundaramoorthy and E. Rotenberg, "A study of slipstream processors," Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000, Monterey, CA, USA, 2000, pp. 269-280, doi: 10.1109/MICRO.2000.898077.

[5] C. Kumar, “Post-Silicon Microarchitecture”, PhD dissertation, North Carolina State University, Raleigh, NC, 2020.

[6] E. Rotenberg, “FoMR: Post silicon micro-architecture, NSF grant no. CCF-1823517,” 2018.

[7] C. Kumar, A. Chaudhary, S. Bhawalkar, U. Mathur, S. Jain, A. Vastrad and E. Rotenberg, "Post-Silicon Microarchitecture," in IEEE Computer Architecture Letters, vol. 19, no. 1, pp. 26-29, 1 Jan.-June 2020, doi: 10.1109/LCA.2020.2978841.

[8] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), October 2002. San Jose, California.

[9] T. C. Mowry, M. S. Lam, and A. Gupta. “Design and Evaluation of a Compiler Algorithm for Prefetching.” In Proc. of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, 1992. DOI: 10.1145/143365.143488. 36

[10] S. Srinath, O. Mutlu, H. Kim and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Scottsdale, AZ, 2007, pp. 63-74, doi: 10.1109/HPCA.2007.346185.

[11] “3rd data prefetching championship.” [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/?final programs

[12] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley and Z. Chishti, "Efficiently prefetching complex address patterns," 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, 2015, pp. 141-152, doi: 10.1145/2830772.2830793.

[13] V. Srinivasan, “Slipstream Processors Revisited: Exploiting Branch Sets”, PhD dissertation, North Carolina State University, Raleigh, NC, 2019.

[14] A. Waterman, Y. Lee, D. A. Patterson, K. Asanovic, Volume I User-level Isa, A. Waterman, Y. Lee, D. Patterson, “The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0”, 2014.

[15] André Seznec. TAGE-SC-L Branch Predictors Again. 5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5), Jun 2016, Seoul, South Korea. (hal-01354253)

[16] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos, "Spatial Memory Streaming," 33rd International Symposium on Computer Architecture (ISCA'06), Boston, MA, 2006, pp. 252-263, doi: 10.1109/ISCA.2006.38.

[17] B. Falsafi and T. F. Wenisch, "A Primer on Hardware Prefetching", Synthesis Lectures on Computer Architecture, vol. 9, no. 1, pp. 1-67, 2014.

[18] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," [1990] Proceedings. The 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, 1990, pp. 364-373, doi: 10.1109/ISCA.1990.134547.

[19] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," Proceedings of 21 International Symposium on Computer Architecture, Chicago, IL, USA, 1994, pp. 24-33, doi: 10.1109/ISCA.1994.288164.

APPENDICES

Appendix A

Additional Details on Core L1D Prefetcher Interface

The prefetch engine code in the core simulator 721sim is a base class that requires that the new prefetcher be a derived class, pass arguments to the C++ base class constructor to initialize the prefetch engine, and an ‘operate()’ function supplied by the specific prefetcher into which relevant information from the prefetch table can be passed as arguments. Beyond this, the prefetcher that is plugged into the prefetch engine can have any structures and features without further constraints.

A.1 Flow of Operation

The following describes the algorithm followed by the prefetch engine:

1. Notify the specific prefetcher about demand access in program order and operate prefetcher.

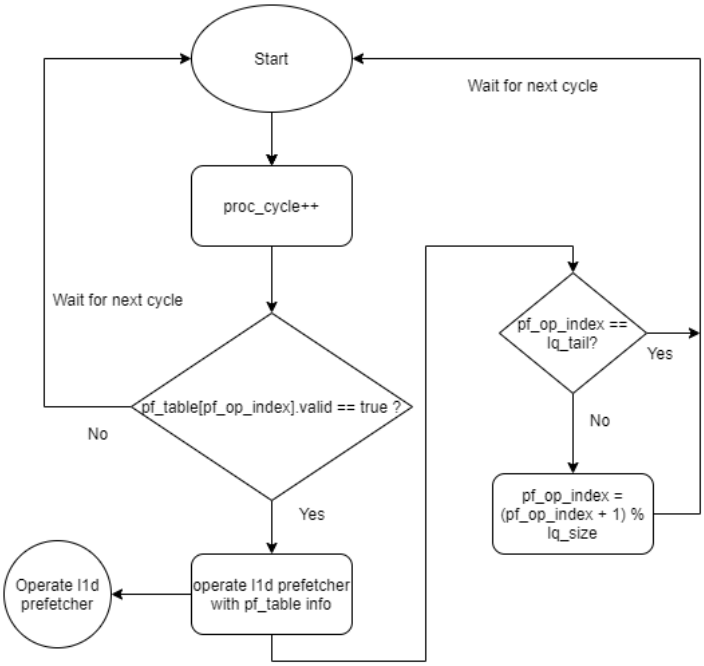


Figure A.1: Flow of operation: Load's notification to the specific prefetcher. Pf_table: prefetch table, lq_tail: load queue's tail

2. Operate the l1d prefetcher and push a prefetch packet to the prefetch queue.

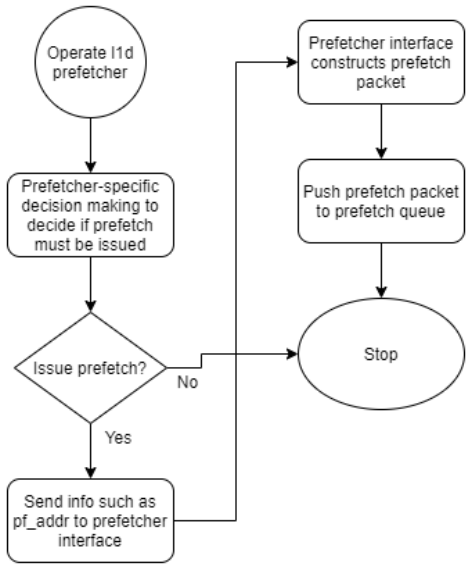


Figure A.2: Flow of operation: Issue prefetch

3. L1D-cache prefetch access if one or more load-store lanes are free (or any other policy of cache-access used such as a dedicated port for the prefetcher)

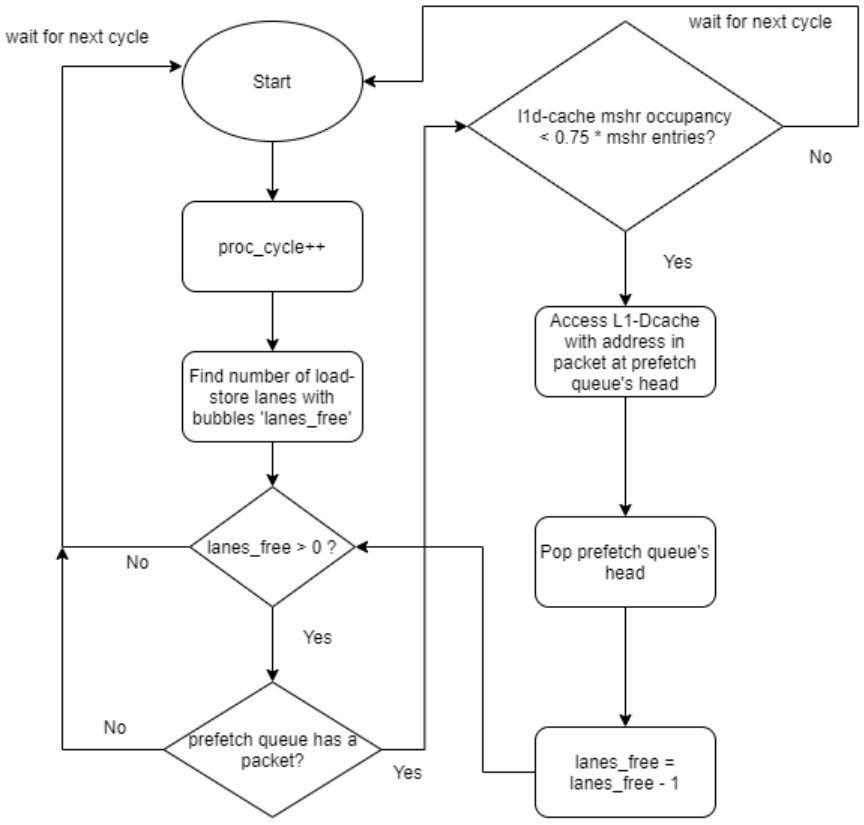


Figure A.3: Flow of operation: Cache access

Appendix B

Algorithm for Stream Buffer Operation

On a cache miss the following occurs:

1. **Read from stream buffer:** If it is a load access (non-prefetch access) or if it is a prefetch access with a switch enabled for the stream buffer to respond to l1d prefetches, then we "read" from the stream buffer, which involves the following steps:
    a. In a loop, for every stream that is valid, we check the head entry of the stream buffer fifo. If the demand address matches with that at the head entry of the fifo:
        i. If the entry is not "available" (indicated by the available bit being reset), we return -1 while setting variables 'hit' and 'bypass' to false. This indicates to the cache that the stream buffer's head entry matched but didn't 'hit' and therefore don't 'bypass' the miss handling to the L1 D-cache.
        ii. If the entry is "available" (available bit is set), we record the stream id that was hit, mark 'hit' to true and break out of the loop. Note that this is the only hit condition.
    b. If no stream was hit, we check for an address match in the unit stride filter:
        i. For every valid filter entry, we check for an address match:
            1. If address matched,
                a. a stream is allocated starting with that address. The process of stream allocation is discussed later.
                b. The entry is deleted and an invalid entry is pushed at the back of the filter queue
    c. If no address matched in the unit stride filter

i. Allocate an entry in the unit stride filter for this address, saved as addr + 1 (a future demand address may directly match against this entry)

ii. Check the non-unit stride filter for a tag match

Note: The tag has been saved as the lineAddr >> 6, where 6 is arbitrary; this is done so that future addresses that lie in the same "spatial map" that is formed by this tag all match up here. This helps detect strides of up to 64.

1. If the tag matches, we check for stride confirmation (done using an FSM state check for the entry, refer to [19] for details)
    a. If the stride has been seen for the first time, save the stride and record this address
    b. If the stride has been seen previously,
        i. allocate a stream with this demand address and stride
        ii. Erase this entry from the filter and push a new invalid entry at the back of the filter queue
2. If the tag did not match, allocate a new entry in the non-unit stride filter

d. If there was no match in either of the filters, no stream allocation took place during the stream buffer read. Therefore, miss handling must be done by the L1 D-cache and so 'bypass' is set to true and 'hit' is set to false

2. **Line allocated but not yet available:** If the stream buffer read returns a -1, it means that the entry is allocated in the stream buffer but not yet available and therefore the cache (and hence the lsu/core) must try again later

3. **Handle stream buffer hit:** Otherwise, if the stream buffer got a hit for this address, handle the stream buffer hit (in a separate function for clarity):
    a. If the evicted line needs a write back, check if L2 and L3 caches have sufficient MHSRs (opportunistic) according to a pre-set limit and check if a free L1 D-cache MHSR is available to accommodate the line from the stream buffer. If yes to all these conditions, then:
        i. Allocate a new line and MHSR for the L1 D-cache for the stream buffer line.
        ii. Replace the old line in the cache.
        iii. The usual flow in CacheClass Access function hereon:
            1. Writeback for dirty line
            2. Saving the correct timestamp for the line to be available in the L1 D-cache. This is saved as the MHSR resolve cycle.
    b. Mark the head entry of the stream to be popped off soon when the stream buffer goes through its usual ‘operate’ cycle
        i. Cycle at which it should be popped off is the same as when the line will be available in the L1 D-cache, fully loaded
    c. If the stream buffer hit handling wasn’t successful (one or more MHSR conditions weren’t met) then L1 D-cache must retry handling the hit later
4. **Bypass Miss Handling to L1 D-cache:** If the stream buffer read returned non-negative and there was no hit in the stream buffer, the usual miss handling flow of the L1 D-cache is followed

Other functions:

5. **Allocate Stream:**
    a. Find the LRU stream and clear it. Update LRU of other streams
    b. Set stream variables (e.g. stride)
    c. Aggressively find as many MHSRs as possible as long as the L2 and L3 MHSR occupancy does not exceed a pre-set limit, stream buffer MHSR consumption does not exceed a pre-set limit and a free stream buffer MHSR is available
    d. Allocate a miss port
    e. Allocate a new cache line structure
    f. Compute the time to load from memory
    g. Add miss latency to access time
    h. Record the next address to be prefetched
6. **Operate stream buffer (every cycle):**
    a. For every valid stream:
        i. Check if the stream's head entry needs to be popped off and the cycle to pop has arrived or passed already
            1. If yes, pop the entry and push an invalid entry at the back of the fifo
        ii. Aggressively free the MHSRs of all the entries whose MHSR resolved cycle has arrived or passed already and mark it 'available' (available bit is set here)
        iii. For the fifo entries that never got an MSHR earlier (e.g. the invalid entry that was pushed at the back of the fifo queue), as long as the MHSR consumption limit (maximum number of MHSRs per stream) is met for the

stream, L2 and L3 MHSR occupancy is within the pre-set limit and a free stream buffer MHSR is available, do the following:

1. Allocate a miss port
2. Access the L2 with the nextAddr as indicated in the stream (previously computed using the recorded stride)
3. Compute access latency
4. Allocate MHSR
5. Make a note of the new address to be prefetched, nextAddr = nextAddr + stride

7. Adaptive Depth selection:

a. Based on the number of stream hits that are recorded in each sampling period, either increment the fifo depth by 8 if performance is increasing or decrement it. The depth saturates at a maximum depth (configurable)