ABSTRACT

SHARMA, SAURABH. Spectral Prediction: A Signals approach to Computer Architecture Prefetching. (Under the direction of Dr. Thomas M. Conte.)

Effective data prefetching requires accurate mechanisms to predict embedded patterns in the miss reference behavior. This dissertation introduces a novel technique *Spectral Prediction* that accurately identifies the pattern by dynamically adjusting to its frequency. The proposed technique exploits the fact that addresses in the reference stream follow definite frequencies and captures them using the recurrence distance information. In so doing, the patterns are successfully detected while the random noise is filtered.

This dissertation describes two implementations of spectral prediction: *Spectral Prefetcher* (SP) and *Differential-only Spectral Prefetcher* (DOSP). The first implementation, SP, is adaptive in behavior and can capture either the pattern of addresses or the pattern of strides between the addresses within the cache miss stream. SP was designed as a proof-of-concept and provided productive insights for designing a more elegant implementation: DOSP, which is resource-efficient and offers better performance. The dissertation also includes simulation driven performance evaluations of SP and DOSP. Our results show that these implementations of spectral prediction achieve 4% to 400% performance improvement for memory-intensive programs running on an aggressive out-of-order processor with large caches and large branch predictor. Additionally, using a set of co-scheduled pairs of benchmarks on a dual-core CMP, we show that a 16KB on chip implementation of DOSP provides an average throughput improvement of 10% and at best by 86%.

Spectral Prediction: A Signals Approach to Computer Architecture Prefetching

by

Saurabh Sharma

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh

2006

Approved By:

Dr. Thomas M. Conte Chair of Advisory Committee	Dr. Greg Byrd
Dr. Purush Iyer	Dr. Eric Rotenberg

कर्मण्येवाधिकारस्ते मा फलेषु कदाचन ।
मा कर्मफलहेतुर्भूः मा ते संगोऽस्त्वकर्मणि ।।

“Freedom is only in the field of action, and not in the field of bringing about the fruits of action. The fruits of work should not be your motive. You should never be inactive.”

Biography

Saurabh Sharma was born on March 16, 1976 in Gwalior, India to Subhash Sharma and Mrs. Rekha Sharma. He obtained his Bachelors degree from Madhav Institute of Technology and Science (MITS) Gwalior, India in December 1998 and his Masters degree in Computer Engineering from North Carolina State University, Raleigh, NC in December 2003. Since January 2003, Saurabh has been pursuing his Ph.D. degree in Computer Engineering at North Carolina State University under the guidance of Dr. Thomas M. Conte on developing architectural techniques to reduce long memory latencies. He worked as an intern at Intel Corporation and Qualcomm from May – December 2003 and February – May 2006, respectively.

Upon graduation, Saurabh is joining the processor development team at Qualcomm, Cary, North Carolina.

Acknowledgements

This wonderful world of graduate school is often frustrating and requires assistance from many people. Here I would like to thank all of those people who were deeply involved in making my Ph.D. journey a successful one. First among these are my dear parents, Subhash and Rekha, who have been the constant source of inspiration throughout my life. I can only hope that I will be as good a parent to my future children as they have been to me. My sister Yuthika and her husband Anurag have always been there when I needed them, and I will always be there for them. My wife, Deepika, also deserves thanks, not only for her love and support, but also for accepting my graduate career with more cheer than I expected. She is my best friend and certainly a far better person than I deserve.

I would like to thank my advisor, Thomas M. Conte for his continuous guidance throughout my doctoral degree. I owe much of my academic and professional development to Tom, who has a unique ability to seek out what is important. His technical insights have been inspiring and his confidence in my abilities has been uplifting. Many of the ideas presented in this dissertation were formed in discussion with Tom. He made his responsibility to make sure that I had enough financial support to accomplish my goals, and sometimes dissuaded me from dropping out like an elder brother. I have been privileged to have Tom as my advisor and mentor, and I hope he knows how much I appreciate everything he has done for me.

I want to thank the other members of my committee: Greg Byrd, Purush Iyer and Eric Rotenberg. Each of them has made invaluable contributions to this dissertation

through discussions and feedback on my writings. In particular, I want to thank Greg and Eric for their constructive criticism and encouragement.

Most of my technical growth has come from interactions with other students rather than with faculty. Fortunately, the TINKER research group has some of the best architects and software designers: Jesse Beu, Paul Bryan, Balaji Iyer, Chad Rosier and Huiyang Zhou. Jesse and Chad deserve special acknowledgement for reading and re-writing my manuscripts, and yet still speaking to me after this horrible experience. Jesse, in particular, has been an intellectual support system as one can ask for. In addition, I want to thank Paul for technical and social discussions that we regularly had while consuming the nicotine delivery device. Lastly, I would like to thank Ahmed-AL-Zawawi, my cube mate for 2 years, for lengthy discussions that we had on every aspect of computer architecture.

I have had many friends in the past six years. One in particular, Surendra Kumar, have patiently listened to my complaints and shared all my academic successes and failures. Our evening coffee sessions at Cup-A-Joe are undoubtedly be my most memorable time of Raleigh. I want to thank Prabhat for being good friend to both Deepika and me.

Finally, I want to thank the administrative staff of the Electrical and Computer Engineering department for taking care of all my paperwork even with my disregard to deadlines and ignorance to some important administrative issues.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

Microarchitectural innovations rely on speculation to boost the performance of contemporary microprocessors. Some of the more effective examples of speculation include branch prediction and data prefetching. Accurate prediction mechanisms have been the driving force behind these innovations, so increasing the accuracy of the predictors increases the performance benefit of speculation. Spectral techniques, such as Fourier analysis, offer the possibility of further improving performance by increasing prediction accuracy [20]. In this dissertation, we show spectral techniques can be implemented in hardware to improve the effectiveness of the data prefetching.

Prefetching is a proactive technique for hiding memory latency and is an essential part of modern microarchitectues [1, 17, 40]. Unlike the demand-fetch models of the caches, prefetching can predict future memory requests and speculatively acquire data. Among all the prefetching techniques, the one that acquire patterns by exploiting the correlation between future memory references with past memory behavior has been shown to be extremely effective [6, 15, 18, 22, 28, 29]. Recent efforts to improve the correlation-based prefetching have primarily focused on alternative structures for storing prefetch history [28, 29]. Although these proposals reduce stale history data, which occurs when history in the distant past no longer represents the current condition, they are

typically associated with low prediction accuracy [29]. We take a different approach – one that is largely orthogonal to previous work – by improving the accuracy of the prediction mechanism itself.

Our work builds on the observation that memory access stream consist of both ordered (pattern) and unordered (random) components. These random components are introduced in the access stream either due to control flow irregularities or due to dynamic transformations (allocation or deletion) of linked-data-structures in an application. We believe that predictions based on these unordered elements (addresses) can result in lower prediction accuracy, which can further lead to higher memory utilization and performance losses. It is natural to ask whether one can improve accuracy by filtering the unordered elements with the help of spectral techniques. Since most spectral techniques would be prohibitively expensive to implement as prefetchers, an alternative is to explore *spectral prediction*, one of the simplest possible spectral technique. Spectral prediction uses the recurrence distance in conjunction with correlation to prune out unordered elements from the access stream. Unlike Fourier analysis, spectral prediction is fairly easy to implement in hardware.

This dissertation is organized into seven chapters. The goal of this chapter is to motivate, introduce and define spectral prediction as well as define its implementations. We begin with a brief summary of the processor-memory performance gap as an obstacle to *instruction level parallelism* (ILP). We continue with a description of the incumbent technique, correlation-based hardware prefetcher, and its limitations. We then introduce spectral prediction as a technique for overcoming the limitations of correlation-based hardware prefetching. Finally, we provide a high level overview of our proposed

implementations of spectral prediction, pointing out their novel aspects. The last section of this chapter summarizes the contribution of this dissertation and outlines the remaining chapters.

1.1 The Problem: Processor-Memory Performance Gap

Processor performance is a product of two factors: clock frequency and instruction level parallelism – the execution of multiple instructions per clock cycle. Significant advancements in semiconductor process technology and microarchitecture have driven both of these factors. Semiconductor technology – the continued miniaturization of CMOS devices – has produced faster and smaller individual transistors, enabling higher clock frequencies. Further, this miniaturization has provided computer architects with more raw materials to design mechanisms for extracting parallelism. These trends have significantly increased processor performance, enabling leaps in software functionality and making processors a ubiquitous commodity.

Even at today's frequencies, processors spend a significant fraction of their time waiting for data from memory. This is mainly due to the diverging rate of improvement between processor technology and that of DRAM memory, i.e. 60% vs. 10% per year [31]. Although the performance of both processor and memory are improving exponentially, the exponent of the processor is substantially larger than that of memory. An inevitable consequence of this difference is a rapidly growing *processor-memory* performance gap, which now is the primary obstacle to improved computer performance.

Because of the growing memory latencies (measured in processor cycles), any request that misses in the cache can eventually take hundreds of cycles to satisfy. As a result, increasing the processor frequency improves performance with diminishing returns as memory latency is not reduced. Moreover, it is often difficult to find useful work in the applications that may be overlapped with memory stalls. Thus to circumvent this problem the prediction of future memory requests is required.

1.2 The Solution: Data Prefetching

To bridge the processor-memory latency gap, computer architects have primarily relied on deeper cache hierarchies, where each level trades off faster lookup speed for larger chip area. Although the use of larger cache hierarchies has proven to be effective in reducing the memory access times for applications showing a high degree of locality in their reference streams, it is still not uncommon for other data sensitive applications to spend more than half of their execution times waiting for memory requests [27]. This poor utilization of caches is partially due to the *demand-fetch* policy of caches, in which data is only fetched into higher levels of cache upon a processor requests. Thus, the first access to a cache line will always result in a miss, since only previously accessed data is stored in the cache. Moreover, if the working set size of an application is large, it is likely that the fetched cache line will be evicted to make room for new lines. When the same line is needed later, the processor must again retrieve it from main memory, once again incurring the full memory access latency.

Many of these misses can be avoided if we augment the *demand-fetch* policy of the cache with a data prefetch operation. Rather than waiting for a cache miss to result in

a memory fetch, data prefetching anticipates such misses and fetches the data before it is required. While several models have been proposed for prefetching either via hardware [6, 9, 11, 15, 18, 19, 22, 28, 29, 30, 33] or software [23, 25, 27], hardware implementations are more popular due to availability of run-time information which can significantly improve the effectiveness of prefetching. Most previous proposals for hardware prefetchers target specific patterns in the access stream – such as strided accesses [9, 19, 30] and accesses to linked-data-structures [11, 33]. While effective for the targeted access patterns, these prefetchers have limited general applicability across a wider range of applications.

1.2.1 The Incumbent: Correlation-based Prefetcher

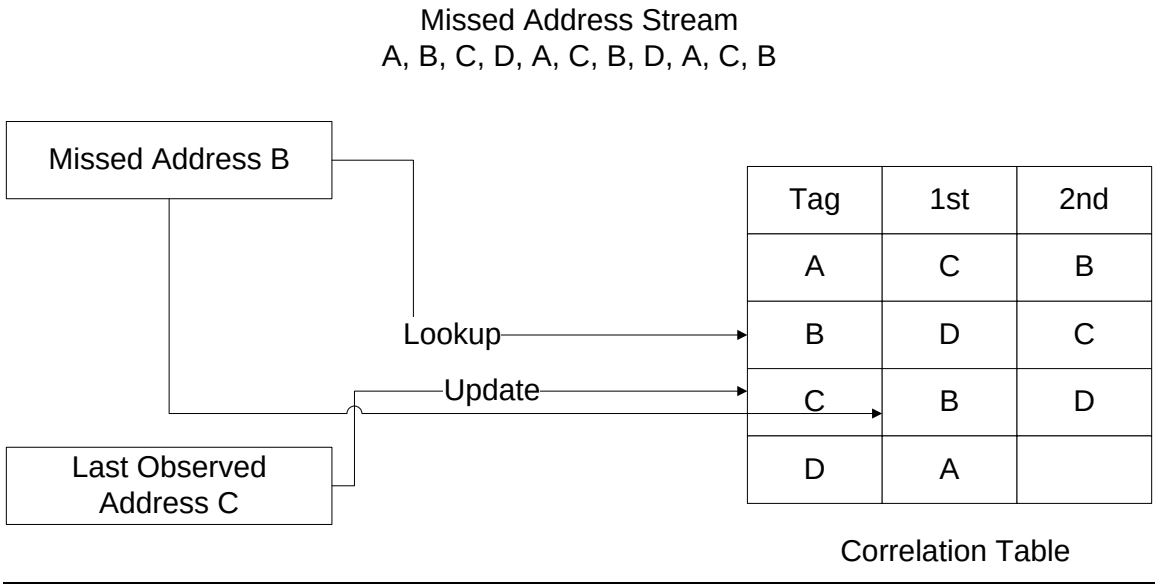


Figure 1-1: An abstract correlation-based prefetcher

Correlation-based prefetching is the standard technique for targeting generalized memory access behavior – including strided accesses, and indirect accesses to linked-data-structures and arrays. The address predictors of correlation-based prefetchers compare future memory references with past memory behavior to prefetch repetitive

reference patterns. These prefetching mechanisms work because programs are repetitive and produce recurring address patterns. An abstract correlation-based prefetcher is shown in Figure 1-1. As shown, the prefetcher uses a history table to record consecutive address pairs. This table is organized as a hardware cache that maintains *parent-children* pair information, where the parent corresponds to the missed cache address and the children correspond to the list of addresses that followed this missed address in the past. Whenever a cache miss occurs, a table entry is accessed and the members of its address list are prefetched. This algorithm can be easily extended from the absolute domain (where patterns of addresses are predicted) to the differential domain (where pattern of strides between addresses are predicted) [28].

1.2.2 Limitation of Correlation-based Prefetchers

Correlation-based prefetchers are not the only mechanisms that correlate values for future predictions. In fact an older more established form of correlation-based prediction is the dynamic branch prediction mechanism [26], where restricted kinds of values – condition bits – are correlated. Although similar in mechanics, these two forms of speculation attack diametrically opposed contexts. The responsibility of the branch predictor is to predict the direction of every conditional branch in the dynamic instruction stream. If a prediction is incorrect, a considerable number of cycles are wasted executing useless instructions and restoring the processor state such that the correct path can be executed. Subsequently, branch predictors are required to use a strict value locality model [24] where every branch outcome is recorded for computing future predictions. In contrast, prefetchers are free from architectural correctness obligations and only serve to

improve the overall performance. Thus, prefetchers can succeed only by intelligently selecting what to prefetch.

The apparent disadvantage of the correlation-based prefetchers is that they are incapable in distinguishing between the repeating pattern and the random noise present in the reference stream. The primary cause for this drawback is adoption of the strict value locality model, which makes the prefetcher abide by the following philosophy: *any given sequence of values will repeat itself*. Thus, a prefetcher learns the requisite pattern on the first pass over the stream and accelerates the handling of the future passes if the pattern remains stable across the passes. However, an irregular pattern that has both repeating and random components can trick correlation-based designs to generate superfluous prefetches. These inaccurate prefetches can be problematic since it causes cache thrashing (eviction of useful data) and consumes bandwidth which may delay demand requests, both of which contribute to increased execution times. Further, with the advent of Chip Multi-Processor architectures [40] where the secondary-level cache and the lower levels of memory hierarchy components are shared, these inaccurate prefetches can have a more detrimental effect due to the additional strain on memory resources.

1.3 What are Unordered Addresses in Memory Reference Stream?

Generally, any recognizable regularity in data is tokenized as a pattern. Here regularity means that the components of the pattern follow the same “order” for different instances of the pattern. For example, modulating a melody (pattern) from C-major to F-major will not change the melody because components of the melody still follow the same order. In

contrast to repeating patterns, *unordered addresses* can be defined as *sequences of values that recur in a random or uncorrelated fashion within a stream*. The following are reasons why unordered addresses exist in the reference stream of an application:

- *Indirect accesses of static or dynamic arrays*: To avoid the wasting of memory, many large scientific algorithms involve sparse matrices [2] which are generally accessed via indirection. Although these encodings save memory, sparse matrix code tends to suffer from poor memory performance due to the use of indirection. Another application that makes use of indirect accesses is event driven microarchitecture simulations, which incorporates dynamic arrays for simulating Reorder Buffers, Load-Store-Queues, etc. The reference stream produced by the accesses of these buffers is irregular and is an artifact of the workload being simulated. These indirect accesses can be easily predicted if they faithfully follow a pattern that remains stable across the whole execution of the program. However, the reference streams of the real world applications, involving indirections, do not always show stable repeating sequences and thus are corrupted by unordered addresses.
- *Transformations of linked-data-structures*: Linked-data-structures (LDS) are dynamically allocated and managed with the help of heap allocation. Elements of the LDS chain contain explicit fields that name all the adjacent elements by addresses. This mode of connectivity allows easy construction and manipulation of data structures of arbitrary shapes, such as trees or graphs. Algorithms involving LDS chains often jump from one part of the tree or graph to another resulting in an irregular access pattern. Moreover, these algorithms often tend to transform LDS chain either by reorganizing their elements (sorting) or by continuous allocation/de-

allocation of elements. Consequently, the access patterns generated by traversing a LDS chain are tainted by the presence of unordered addresses.

- *Control flow irregularities in applications*: Control flow constructs in high level languages such as conditional statements is another source for inserting unordered addresses in the reference stream. The outcome of these conditional statements (branches) depends on the context in which they are executing and are often predictable. For instance, a branch may always be taken in one sharing phase and not taken in another. Problems arise when some branches produce patterns which externally do not appear regular or repetitive (e.g. a branch that tests a data element obtained form external input). As a result, the paths guarded by these branches will be executed in an unpredictable manner and the references to objects in their respective paths will appear as unordered addresses in the access stream.

To get a clear picture of the unordered addresses, we provide an example in Figure 1-2 that includes both a repeating pattern and unordered addresses. The graphs present the miss access pattern of the *mcf* benchmark. The first graph shows the repeating pattern while the second and the third graphs depict the partial snapshots of the repeating pattern. The unordered addresses are symbolized by the crosses in grey while the dots in red color represent addresses that constitute a pattern in the reference stream. We will follow this convention in rest of the dissertation. Our convention is to use arrows to represent recurring distances of the repeating addresses.

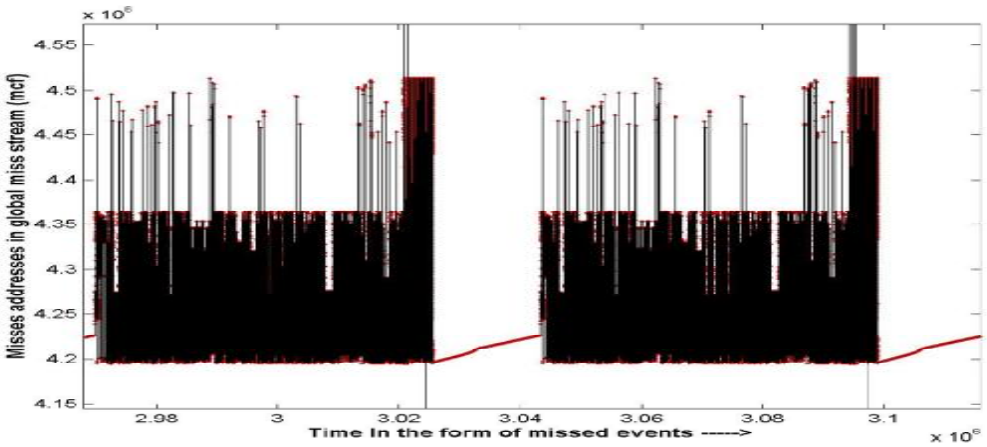


Figure 1-2 (a)

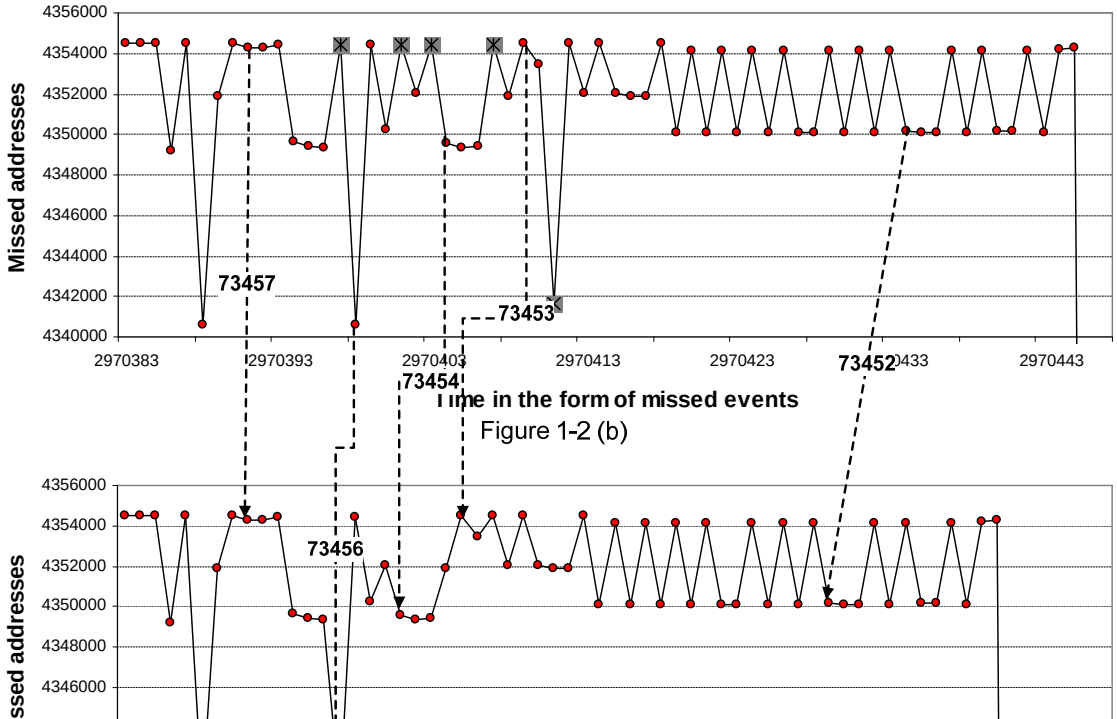


Figure 1-2 (b)

Figure 1-2 (c)

Figure 1-2: Miss access pattern of *mcf* benchmark

There are two important implications of these graphs. First, most of the addresses reappear by five different recurring distances: *73452, 73453, 73454, 73456* and *73457*.

As the horizontal axes of these graphs represent “time” measured in successive missed references, we can safely state that components of the pattern are reappearing with the following frequencies: *1/73452, 1/73453, 1/73454, 1/73456* and *1/73457*. Second, frequency variations among the reappearing addresses are due to the presence of unordered addresses. Thus, if there were no unordered elements present in the reference stream, the addresses would have reappeared with the same recurring distance.

1.3.1 Impact of Unordered Addresses on Caches and Prefetchers

The impact of unordered addresses on the memory hierarchy (caches) and on proactive management of the memory hierarchy (prefetching) is widely divergent. To illustrate this issue, the impact on caches and prefetching are discussed separately. Caches do not use speculation: data is fetched as the processor demands it. So the only problem that unordered address can pose is the eviction of useful data. Moreover, if size is not an issue, caches can acquire any reference stream – with or without irregular behavior. On the other hand, prefetchers use speculation, and can either predict an unordered address or can make predictions based on these unordered addresses, both resulting in ineffectual prefetches. Unlike caches, a larger table size does not affect the prediction accuracy of the prefetcher. Thus in order to increase the effectiveness of the prefetchers, we need to seek out new mechanisms that can filter the unordered addresses and acquire only the repeating patterns for predictions.

One plausible solution for filtering the unordered addresses is the use of Fourier analysis, which can transform the address history information from the time domain to the frequency domain. This transformation generates spikes for frequencies with which the components of the pattern are reappearing in the time domain. Further, these

frequencies can be passed to a narrow bandpass filter that isolates parts of the time domain signal which gave birth to the spikes in the frequency domain. In this manner pattern can be isolated from the unordered addresses, as they will have none or negligible frequency amplitude in the frequency spectrum. Although this solution is unrealizable in terms of hardware implementation it motivates us to recognize that there is still room for improvement. We believe that there are many more ideas yet to be discovered in the field of data prefetching. Indeed, we introduce one such technique in the next section.

1.4 Spectral Prediction

The limitations of correlation-based prefetchers have lead to the following hypothesis: prefetchers should only acquire repeating patterns for predictions. However, this leaves us with two requirements. First, we must devise a simple mechanism that can isolate the repeating patterns from the unordered addresses. Second, the isolation mechanism should be dynamic (i.e. it should take into account the changing conditions at run-time) since addresses identified as unordered in one phase may become a part of the pattern in another. The observation that motivates for such a filtering mechanism is that the elements (addresses) of the pattern collectively reappear with the same recurring distance in the reference stream (as shown in Figure 1-2). Provided that the filtering mechanism records the position of values in the stream and has the ability to identify values that share the same recurring distance, patterns can be reliably detected and acquired from the stream. We call this kind of acquisition and subsequent prediction of patterns as *spectral prediction*. Figure 1-3 shows a simplified flow diagram governing the pattern detection algorithm of spectral prediction.

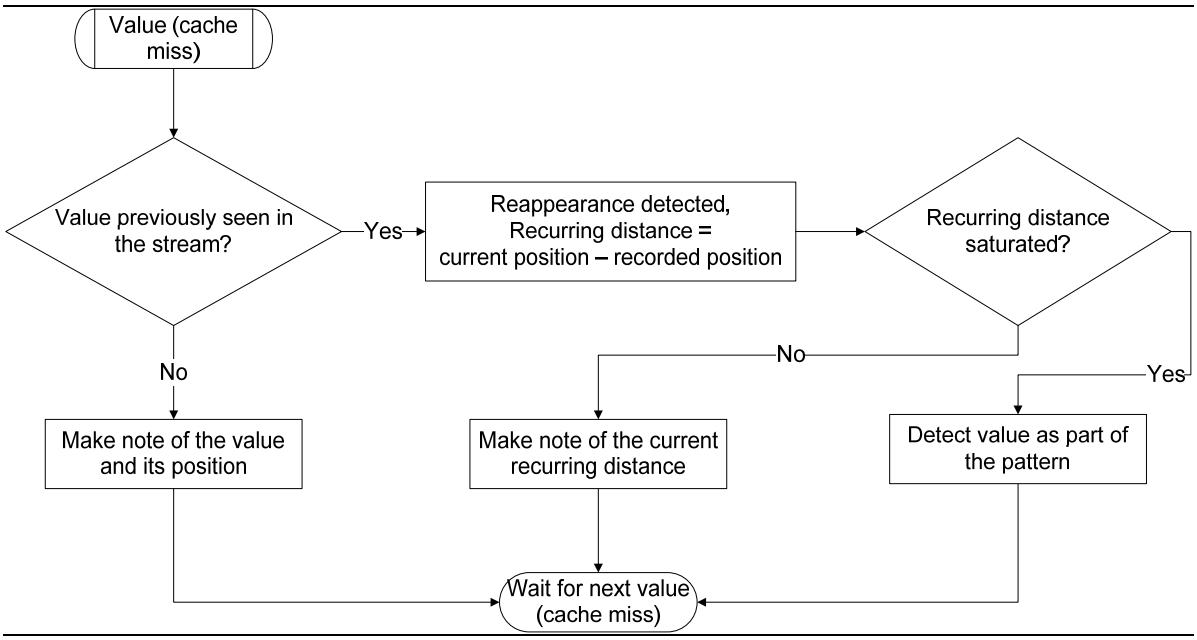


Figure 1-3: Simplified flow diagram for spectral prediction

This algorithm assumes that the filter has two buffers. The first buffer records values as well as their positions in the stream while the second buffer records the recently observed recurring distances with which the values are reappearing in the stream. The algorithm also assumes that after observing a recurring distance a certain number of times the corresponding entry in the second buffer becomes saturated. As shown in the flow diagram, for a value to be validated as part of the pattern, it must arrive with a recurring distance that has been saturated. Since the recurring distance of a random or unordered value will never saturate, it will be easily discarded.

This dissertation explores the use of spectral prediction as an augmentation to correlation-based prefetching. We propose two novel prefetching mechanisms that implement spectral prediction for pattern detection and prediction within a stream. The address predictor of the proposed mechanisms is based on a simple idea – *predict only*

the repeating contexts and leave the management of random noise with the demand fetch model.

1.4.1 Aspects of Spectral Prediction

To help understand the important aspects of spectral prediction we provide a formal definition. Spectral prediction is characterized by two properties:

- *Dynamically adjusts to the frequency of the pattern*: To understand this aspect, we take an example of a very simple kind of pattern – *sine wave*. Every point in this wave "collectively" reappears with the same recurring distance called the *wavelength*. So a predictor based on spectral prediction will quickly recognize this fact and will detect all the elements of the wave as part of the pattern. Spectral prediction, as proposed, exploits this property – collective reappearance – for pattern detection and in a way tunes to the frequency (or frequencies) of the pattern.

- *Applicability to any correlation-based method*: The strata required for spectral prediction is already present in the correlation based methods, such as ordered and up-to-date history. The modification required is to provide *know-how* for detecting the recurring distances with which the correlated pairs reappear in the stream. Therefore, spectral prediction can be applied to any correlation-based address prediction mechanism – absolute or differential.

1.4.2 Implementation

As previously noted, a spectral prediction implementation has three components. The first component is a *pattern history buffer* that records the previously seen correlated pairs in the reference stream. Similar to a correlation-based design, this buffer can be

organized as a set associative table. In addition to the correlated pairs this buffer also stores the position of the pair in the reference stream. The second component, the *position counter*, maintains the continuous count of the miss events. This counter provides the position of the pair in the stream and is used for calculating the recurring distances of the reappearing pairs. The final component is the *recurrence distance history buffer* that maintains the previously observed recurring distances with which the values (addresses) are reappearing in the stream. This buffer is used for detecting pattern in the reference stream.

This dissertation proposes two prefetching architectures that implements spectral prediction by using the above defined components. Both these implementations prefetch for the lowest level data cache (in our case the L2) because modern out-of-order processors can tolerate L1 data cache misses with little performance degradation. Our first solution, the *Spectral Prefetcher* (SP), is an adaptive approach that detects either patterns of addresses (absolute patterns) or patterns of strides between the addresses (differential patterns) within the L2 cache miss stream. The prefetcher dynamically determines whether the absolute or the differential domain will increase the effectiveness of prefetching and switches accordingly. This first implementation acts as a proof-of-concept and uses a rather complex algorithm for implementing spectral prediction. Our second solution, the *Differential-only Spectral Prefetcher* (DOSP), detects only stride patterns in the global miss stream. DOSP requires significantly smaller hardware structures and provides better performance improvements while employing a simplified algorithm for implementing spectral prediction.

This dissertation includes a simulation-driven performance evaluation of SP and DOSP. The baseline for this evaluation is an aggressive 8-wide superscalar processor with larger caches and a larger branch predictor. Despite high baseline performance, these implementations of spectral prediction achieve 4% to 400% speedup on memory intensive benchmarks. Additionally, using a set of co-scheduled pairs of benchmarks on a dual-core CMP, we show that a 16KB on chip implementation of DOSP provides an average throughput improvement of 10% and at best 86%. Performance improves as both SP and DOSP successfully reduce the average load latency in these programs.

1.5 Dissertation Contributions and Outline

This dissertation's thesis comprises of the following two assertions:

- ***Spectral prediction is a good method for increasing the effectiveness of correlated prefetchers.***
- ***Spectral prefetchers are good instantiations of spectral prediction that maintain accurate identification of a pattern by adjusting to its frequency.***

In this dissertation I make the following contributions:

- I introduce and define spectral prediction.
- I introduce and describe two forms of Spectral prefetchers, implementation of spectral prediction.
- I present an empirical evaluation of Spectral prefetchers in support of my thesis.

The remainder of this dissertation is organized into six chapters. In Chapter 2, we describe our experimental framework. In Chapter 3, we present an empirical analysis of the spectral prediction with the help of probabilistic autocorrelation technique. Chapter 4 presents the first instantiation of spectral prediction: *Spectral prefetcher* (SP) while Chapter 5 presents the second instantiation of the spectral prediction: *Differential-only Spectral Prefetcher* (DOSP). This chapter contains an empirical, simulation-driven evaluation of DOSP. In this chapter, we also discuss the limitations of the SP. In Chapter 6 we discuss the related work which is followed by conclusion in Chapter 7.

Chapter 2 Methodology

In this chapter, we explain the general methodology we use to obtain our experimental results. Later sections will go into more detail where appropriate. There are two main types of results that we gather: instructions-per-cycle (IPC) and metrics measuring the "goodness" of the prefetcher. We gather these statistics in context of an out-of-order simulator, described in the next section. Then in Section 2.2 we describe our suite of benchmarks and present some of their execution characteristics. Finally, in Section 2.3, we provide a brief review of two implemented correlation-based prefetchers that we used in this dissertation for comparing with our schemes.

2.1 Simulation Environment

The simulator used in this thesis is comprised of two different simulators: *functional* and *timing* simulator. Both of these simulators are written in C++, using several components from the publicly available Simplescalar toolset [4]. These simulators interpret an instruction set derived from the MIPS-1 ISA [21]. The fast functional simulator reads and executes the instructions from the program binary, without modeling any microarchitectural details and execution times. The primary utility of the functional simulator is to verify the timing simulator on-the-fly and for skipping the initial parts of

the benchmarks. On the other hand, timing simulator models the microarchitectural behavior of an out-of-order processor at the cycle-by-cycle level. In this section, we present the microarchitectural details of our timing simulator as well as the basic prefetching environment assumed for the prefetcher implementation. In addition, we briefly discuss how we calculate the performance metrics.

2.1.1 Microarchitectural Details

The timing simulator models an execution driven dual-core Chip Multiprocessor that executes only the user level instructions. Each core is an out-of-order superscalar processor that resembles a popular microarchitectural style – *physical register file* – used by the MIPS R10000 [42] and Pentium 4 [16]. In this style, a centralized register file is used for all the register state – speculative and non-speculative – and the values are referred by a physical register number. Some important characteristics of the timing simulator include:

- The simulator models the base pipeline faithfully i.e. the micro-operations (*Fetch, Decode, Rename* etc.) are not faked rather they are performed in the appropriate pipeline-stages.
- The simulator is *execution-driven* and moves down any speculative path until the miss-prediction is detected. In case of miss-prediction, recovery is performed by flushing the pipeline.
- The execution model of the pipeline and the memory system are event-driven. A central data structure is used to maintain an event list consisting of events that are scheduled for the future execution in simulation time. Our simulator does not perform an action for every cycle rather it invokes an action for a module at the scheduled

time. Similarly, an event list is maintained for memory system events that invokes actions – writing requested cache lines, allocation/deallocation of miss-status handling registers (MSHR) etc. at the proper time.

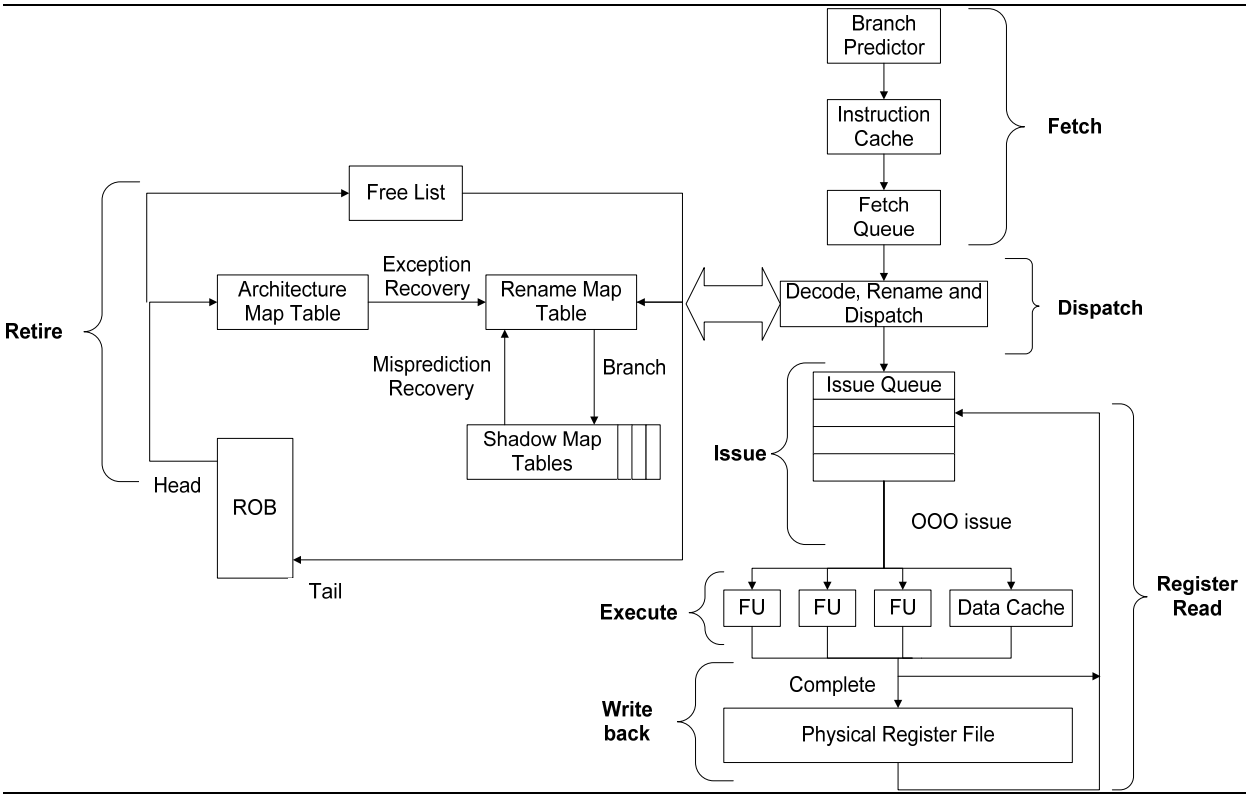


Figure 2-1: Microarchitecture modeled for each core

The pipeline and the microarchitecture of the single core are shown in Figure 2-1. Each core has a private L1 instruction and data caches while the L2 and all lower levels of memory hierarchy are shared between the two cores (not shown). The pipeline consists of seven stages: *Fetch, Dispatch, Register Read, Issue, Execute, Writeback* and *Retire*. All stages except the execute stage are a single cycle in length. The execute stage is of variable length, depending upon the latency of the executing instruction. Next, we describe the various microarchitectural operations performed for processing an instruction.

The instruction *fetch* stage reads instruction(s) from the instruction cache. The maximum number of instructions brought into the processor per cycle is a configurable parameter. The fetch unit also uses branch prediction to fetch across control transfer instructions. The instructions are read from the fetch queue during the instruction *dispatch* stage. This stage decodes the instruction and renames their operands; it also allocates entries for the instructions in the *Reorder Buffer* and the *Issue Queue*. In the *Register Read* stage, the register operand values are read from the register file depending upon the *architectural map table* or the *rename map table*, which ever contains the latest mapping of the register. If an operand value is not ready, a tag (ROB index) identifying its producer is stored in the corresponding entry of the instruction queue. Whenever the producer instruction completes, the corresponding tag and value are broadcast to the *wakeup-select* logic, which awakes the waiting instruction(s). The *issue* stage selects and dispatches the ready instructions from the issue queue to the functional units for execution. For load instructions, the dispatching (accessing of the data cache) takes place only when there are no pending store instructions with unknown address. If a load address matches the address of a store ahead in the pipeline, the value is *forwarded*. After an instruction completes its execution, its results are written to the register file and the corresponding ROB entry is broadcast into the wakeup-select logic. Finally, the retirement takes place when the instruction reaches at the ROB head and the architectural state of the machine (architectural map table and memory) are updated.

The configuration of the performance simulator is detailed in two tables. Table 2-1 shows the pipeline characterstics while Table 2-2 shows the branch predictor and

memory hierarchy parameters. The base configuration of each core is an aggressive, 8-wide processor with large caches and branch predictor tables.

Table 2-1: Pipelining and scheduler configuration of each core

Dual-Core with 2 GHz Core Frequency		
Parameter		**Configuration**
Fetch	Width	A maximum of 8 instructions are fetched
	Queue	16
	Latency	1 cycle
Decode	Width	A maximum of 8 instructions are decoded
	Latency	1 cycle
Rename	Width	A maximum of 8 instructions are renamed
	Physical Registers	160 = 32 Logical registers + 128 in-flight instructions
	Latency	1 cycle
Retire	Width	A maximum of 8 instructions are retired
Buffers	Reorder Buffer	128 entries
	Load-Store-Queue	64 entries
Functional Units		8 fully symmetric functional units, Integer ALU operations takes one cycle while complex operations have MIPS R10000 latencies
Scheduler	Instruction Queue	64 entries
	Disambiguation	Loads stall when there is a pending store with unresolved address
	Register Read Latency	1 cycle
	Address generation Latency	1 cycle
	Store Forward Latency	1 cycle
	Scheduling around cache misses	In case of cache miss, event is created that access the next level of memory hierarchy. Scheduler snoops the corresponding MSHR in order to get actual resolved cycle of the missed load instruction

Table 2-2: Branch predictor and memory hierarchy of the simulator

Branch Predictor and Memory Hierarchy		
Parameter		**Configuration**
Branch Predictor (private in each core)	Conditional Branches	64K entry Gshare
	Taken Targets	32 K entry BTB
	Return Addresses	1024 entry RAS
Memory Hierarchy	L1 Instruction Cache (private)	4-way 64KB, 64 byte line size, with LRU replacement policy and 1 cycle access latency
	L1 Data Cache (private)	4-way, 32 KB, 64 -byte line size, write back write allocate, with LRU replacement policy and 1 cycle access latency
	L1D MSHRs (private)	8
	L2 Unified Cache	8-way, 2 MB, 64 -byte line size, write back write allocate, with LRU replacement policy and 10 cycle access latency
	L2 MSHRs	8
	L1/L2 Bus	32 byte wide operates on 1 GHz frequency
	L2/Memory Bus	8 byte wide operates on 512 MHz frequency
	Memory Latency	200 cycles

2.1.2 Prefetching Environment and Architectural Modification

Our proposed framework assumes that the prefetching methods inspect the L2 miss stream and prefetch directly into the L2 cache, as shown in Figure 2-2. We prefer the prefetcher to be an integral part of the memory hierarchy (i.e. on-chip) rather than as a part of memory controller (i.e. off-chip). Placing the prefetcher on-chip is preferred because it provides ready access to the L2 cache and to the L2 MSHR. In our simulation environment, before issuing a prefetch to memory, both the L2 cache tag array and MSHR are probed to see if the prefetch address is present in the cache or currently in flight. If the address is found, the prefetch request is dropped. We also ensure that the

MSHRs are shared between L2 prefetch and demand requests. If in the process of issuing a prefetch request no free MSHR is available, the prefetch request is squashed. Once a prefetch is issued, a MSHR is allocated to bring the corresponding cache line into the L2 cache. In the event that a demand request encounters an in flight prefetch request (i.e. when there is a tag hit in the MSHR but the line is not present in the cache) for the same cache line address, the demand request waits until the cache line is written into the L2 cache.

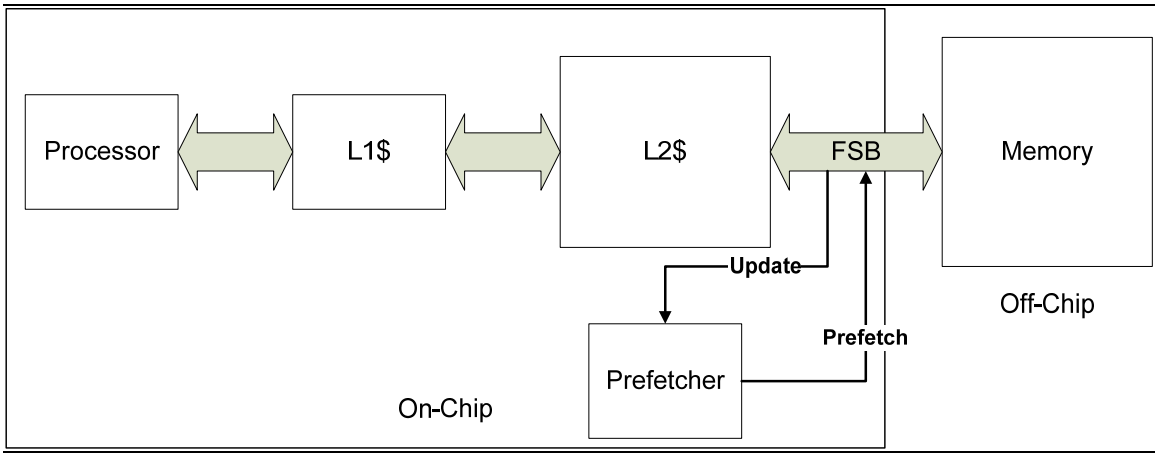


Figure 2-2: Proposed prefetching microarchitecture

In order to maintain the "natural" L2 demand miss stream, we propose one-bit prefetch flags be added to the L2 cache lines and L2 MSHRs. This is done to distinguish between a demand requests and a prefetch requests. When a prefetch request is issued, the prefetch flag of the allocated MSHR is set. Similarly, when a prefetched line is written into the L2 cache, its prefetch flag is set. Whenever the demand request either hits on a prefetched line or has a partial-hit on an L2 MSHRs allocated by a prefetch request, the flag is cleared and the address is sent to the prefetching hardware as if it were an L2 cache miss.

2.1.3 Performance Metrics Calculation

As previously mentioned, we present two main types of results: IPC and prefetching metrics. For IPC measurements, we measure the number of cycles and instructions executed for a particular benchmark. We then divide the number of retired instructions by the number of executed cycles. In chapter 4 and 5, we typically use the average IPC for providing limit study analysis results. We calculate average IPC by taking the harmonic mean of the individual IPCs of all the benchmarks.

We also provide results for three important metrics traditionally used for comparing prefetchers: *coverage*, *accuracy* and *timeliness*. Here coverage indicates the fraction of memory requests that were supplied by the prefetcher while accuracy indicates the fraction of prefetched cache line that were actually used by the processor [18]. The coverage and accuracy results were calculated using the following formulae:

$$\text{cov}\,erage = \frac{misses\ without\ \ prefetching - misses\ with\ \ prefetching}{misses\ without\ \ prefetching}$$

$$accuracy = \frac{misses\ without\ \ prefetching - misses\ with\ \ prefetching}{number\ of\ \ prefetches\ generated}$$

When calculating the performance results, coverage and accuracy are not adequate metrics since they do not provide any information about the criticality of the load misses being masked by the prefetchers. As a result, we also provide results for timeliness, which indicates if the data provided by the prefetcher arrives before it is needed. Since there is no definite procedure for measuring timeliness, we define our own method, where demand requests served by the prefetcher are broadly categorized into three categories. We call prefetch requests as *timely* if they are present in the cache while serving the demand requests or serve demand requests within a quarter of memory

latency. Demand requests serviced within half of the memory latency are called *acceptable* and finally all others are termed as *poor*.

2.2 Benchmark Programs

The benchmark set that we use in this thesis consists of 10 programs: 5 SPEC CPU2000 integer programs [38], 4 SPEC CPU2000 floating-point programs and a micro-benchmark from the pointer intensive Olden suite [5]. All SPEC benchmarks use the *reference* input sets while the micro-benchmark uses a synthetic input. Since the micro-benchmark – *mst* – is not as well known as those in SPEC benchmark suite, we describe it further here. *Mst* computes the minimum spanning tree of the a random graph. It has a single dominant computational loop that iterates over the unattached nodes and finds the node with the shortest edge to one of the nodes already present in the graph. Finally, the selected node is attached to the graph.

All C benchmarks were compiled using the (*gcc-based* version 2.6.3) simplescalar compiler with the following optimization flags: -O3, –funroll-loops and -finline-functions. The FORTRAN benchmarks were first converted to C using the *f2c* program and then compiled using the simplescalar compiler. In the interest of reduced simulation time, we simulate 100 million instructions for each SPEC program, skipping to the first SimPoint [36]. For *mst* 100 million instructions are skipped and then the next 100 million instructions are executed. We also present results for a dual-core CMP that enable L2 cache prefetching. For these experiments, program pairs are co-scheduled to run on separate CMP cores and each pair is skipped to the first SimPoint (except *mst*). The

program pairs are executed for 100 million core cycles, and hence, the benchmark with the higher IPC will execute more instructions

Table 2-3: Functional benchmark characterization

Memory Intensive Benchmarks				
Benchmarks	**Branch Prediction Rate**	**L2 Cache Misses**	**Base IPC**	**IPC with Oracle L2**
ammp	99.83%	99.99%	0.04	0.39
mcf	95.57%	52.45%	0.12	0.37
mgrid	99.12%	60.14%	0.89	2.69
mst	99.73%	61.63%	0.43	1.56
swim	99.68%	46.70%	0.52	2.26
Compute Intensive Benchmarks				
Benchmarks	**Branch Prediction Rate**	**L2 Cache Misses**	**Base IPC**	**IPC with Oracle L2**
art	99.43%	0.00%	0.79	0.79
gcc	95.18%	0.02%	1.45	1.58
parser	95.71%	7.81%	1.17	1.45
twolf	90.52%	0.01%	1.19	1.24
vpr	93.20%	0.00%	1.41	1.47

In Table 2-3, we present four baseline results for each benchmark to show their relative characteristics. These results are branch prediction rate, L2 cache miss rate, base IPC and IPC improvement with an oracle L2 cache. These results were obtained using the baseline processor described in this chapter with the configuration shown in Table 2-1 and Table 2-2. As shown, the benchmarks fall into two categories. At one extreme, *ammp, mcf, mgrid, mst* and *swim* spend a significant fraction of their execution times

waiting for memory requests. We classify these benchmarks as *memory intensive.* At the other extreme, *art, gcc, parser, twolf* and *vpr* rarely stall on memory requests since their working sets completely fit in the L2 cache of our simulation environment. These benchmarks are classified as *compute intensive*. The selection of these benchmarks allow us to study three different scenarios for CMP architectures that enable L2 cache prefetching: i) when two memory intensive benchmarks are co-scheduled, ii) when a memory intensive is co-scheduled with a compute intensive, and iii) when a pair of compute intensive benchmarks are co-scheduled. These scenarios provide insight into the varying impact a prefetcher can have in a CMP environment. For example, a prefetcher may degrade the performance of co-scheduled compute intensive benchmarks by disturbing cache locality while an aggressive prefetching mechanism may help a pair of memory-intensive benchmarks. In Chapter 5, we show that an accurate prefetching mechanism can improve performance in majority of these scenarios.

2.3 Prior Correlation-based Prefetching Mechanisms

There are countless proposals for hardware and software data prefetching. We will briefly describe some of them in Chapter 6. In this section we will focus on recently proposed prefetching architectures that we compare against our implementations.

2.3.1 Prefetching with a Global History Buffer

Most correlated prefetchers use a set associative prefetch history table that is directly accessed using an index value. In Global History Buffer (GHB) based prefetchers [28], however, the history is held in an *n*-entry FIFO table called the Global History Buffer. Figure 2-3 depicts the structure of a GHB based prefetcher where the table holds

the most recent L2 cache line miss addresses. Each entry of the GHB also stores a link pointer to maintain a time-ordered linked list of previous entries with the same index key. The Index Table holds the head pointer to these linked lists and is accessed via some key. Depending on the key that is used, many prefetching algorithms can be implemented.

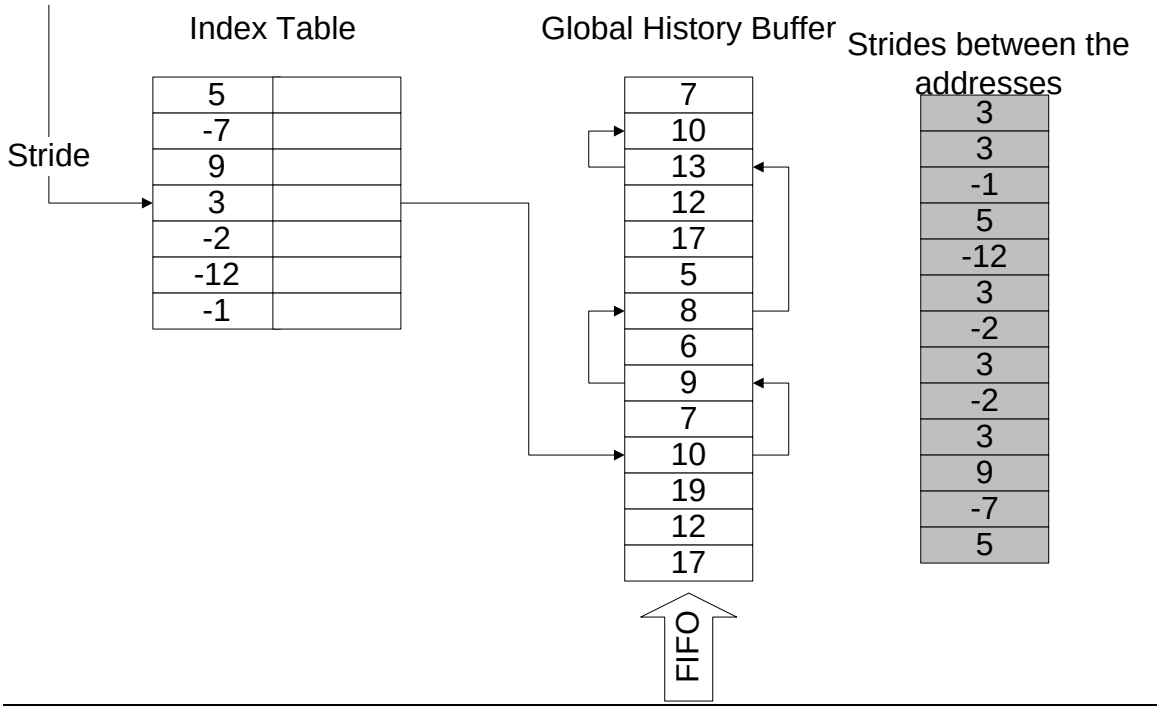


Figure 2-3: Global History Buffer implementing Differential Markov

Figure 2-3 also shows an example of Differential Markov or Distance Prefetching [28] algorithm using a GHB. The Index Table is accessed via a stride between two consecutive cache line addresses. This in turn points to the most recent occurrence of the address that generated the same stride. As shown in Figure 2-3, strides are extracted by finding the differences between the addresses – the strides themselves are not stored. In this dissertation, we present results for GHB Differential Markov width-prefetching and GHB Differential Markov depth-prefetching. In width prefetching, the strides adjacent to each linked list element are separately added to the miss address to generate multiple

prefetch addresses. For depth prefetching, addresses are generated by cumulatively adding strides to the miss address beginning with the head of the linked list and progressing sequentially toward the head of the GHB.

2.3.2 Tag Correlating Prefetcher

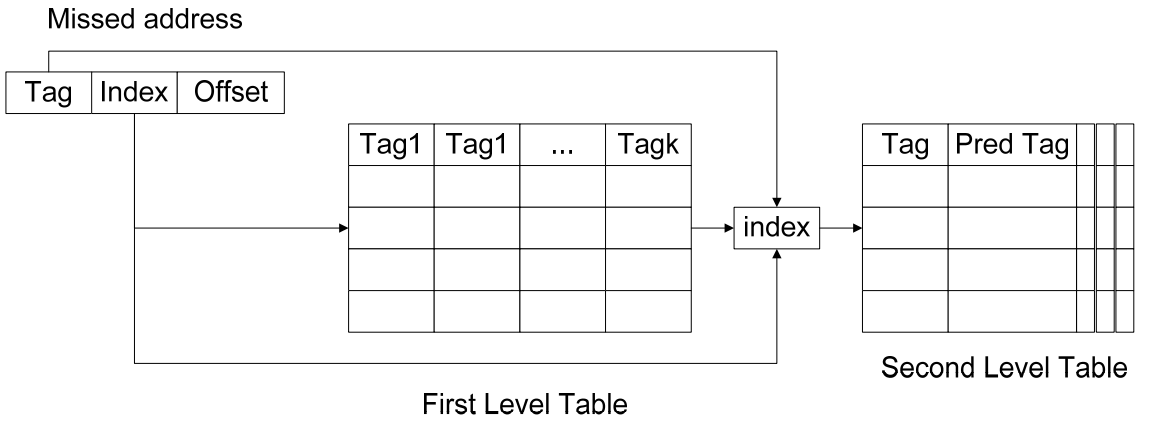


Figure 2-4: Structure of Tag Correlating Prefetcher

Tag Correlating Prefetcher (TCP) [15] predicts the pattern of tags for a given cache set. Figure 2-4 depicts the structure of TCP. It is organized as a two-level correlating prefetcher where the first level table contains the last observed miss tags of the same cache set. These tags are combined together with the current miss tag and cache set to access the second level table. The second level table in turn provides the next predicted tag, which is combined with cache set to generate a prefetch address.

Chapter 3 Miss Spectrum Analysis

In Chapter 1, we advocated the use of spectral prediction as a new paradigm for improving the effectiveness of prefetching. Before we can discuss the methods for implementing spectral prediction, we need to develop a better understanding of the reference behavior of applications. We should be aware of various characteristics – such as the presence of pattern(s) in the reference stream, or the occurrence of random addresses and their potential effects on the repeating patterns. Only after gaining such an understanding, we will be able to design an effective implementation.

To achieve this goal, we need to identify pattern characteristics within the reference streams of various applications However, this leaves us with the dilemma as to what sort of analysis should be conducted. Fortunately, there are several *data analysis* techniques [7, 13] that can be used for revealing the nature of the reference streams. In this chapter, we use one such technique – *probabilistic autocorrelation* – to detect and uncover the underlying patterns present in the reference stream. We begin by first briefly describing autocorrelation providing the mathematical background with the help of few examples. Then in section 3.2 and 3.3, we will discuss the autocorrelation plots for the reference stream of benchmarks that are used in this thesis. After gaining insight from the

autocorrelation plots, we present arguments for the predictor working in both the absolute and differential domains.

3.1 Autocorrelation: A Spectral Method for Detecting Patterns

Autocorrelation plots are a commonly used tool for checking randomness in a data set. This randomness is ascertained by computing autocorrelation coefficients for data values at varying time lags. Here lag is a fixed time displacement. For example, we can calculate autocorrelation coefficient between the values of same data set at times i and $i + k$, where k is called the lag. If the data set is random, autocorrelation coefficients should be near zero for any and all time-lag separations. If non-random, then one or more of the coefficients will be significantly non-zero.

For completeness, there is need for an in depth review of the mathematics and techniques associated with autocorrelation. In the next sub-sections, we present the mathematical background as well as the properties of the autocorrelation coefficients.

3.1.1 Mathematical Background

The autocorrelation coefficient can be thought of as a measure of how similar a data set is to a time shifted version of itself. The formula for calculating autocorrelation coefficient r_k for any data set Y is:

$$r_k = \frac{\sum_{i=1}^{N-k} (y_i - m_y)(y_{i+k} - m_y)}{\sum_{i=1}^{N} (y_i - m_y)^2} \qquad (3.1)$$

Here N is the size and m_y is the mean of the data set $Y = y_1, y_2, y_3 \ldots y_N$ while k is the amount the data set has been shifted, usually referred to as *lag*. The values of the coefficients lie between -1 and 1, exclusively. A high absolute value of coefficient at a given lag indicates a close relationship while a low value indicates a less definite relationship.

Another useful way of representing the autocorrelation coefficient formula is in terms of the *expected value.* In probability theory, the expected value is the sum of the probability of each possible outcome multiplied by its value: it represents the average amount one "expects" to win per bet if bets with identical odds are repeated many times. In terms of data set where each element appears with the same probability, we can safely say that the expected value will be equal to the mean of the data set. The autocorrelation coefficient in terms of expected value can be written as:

$$r_k = \frac{\left\langle (Y - m_y)\,(Y_k - m_y) \right\rangle}{\left\langle (Y - m_y)^2 \right\rangle} \tag{3.2}$$

Here the $\langle\ \rangle$ symbol represents the expectation operator and m_y is the mean of the data set Y. The data set Y_k is obtained by time shifting the original data set by a lag of k. We also assume that the original data set is so large that creating a new data set by shifting with a given lag does not change its mean. Equation 3.2 can be further reduced as following:

$$r_k = \frac{\left\langle (YY_k - m_y Y_k - m_y Y + m_y{}^2) \right\rangle}{\left\langle (Y^2 - 2m_y Y + m_y{}^2) \right\rangle} = \frac{(\left\langle YY_k \right\rangle - m_y \left\langle Y_k \right\rangle - m_y \left\langle Y \right\rangle + m_y{}^2)}{(\left\langle Y^2 \right\rangle - 2m_y \left\langle Y \right\rangle + m_y{}^2)}$$

$$= \frac{\left\langle YY_k \right\rangle - m_y{}^2}{\left\langle Y^2 \right\rangle - m_y{}^2} \tag{3.3}$$

A reader familiar with statistical theory can quickly recognize that autocorrelation coefficient, as defined in equation 3.3, represents *autocovariance* ($\langle YY_k \rangle - m_y{}^2$) when normalized by *variance* ($\langle Y^2 \rangle - m_y{}^2$). We will use this equation in explaining some of the properties associated with autocorrelation in the next sub-section.

3.1.2 Behavior of the Autocorrelation Coefficients

In order to understand the behavior of autocorrelation coefficients, we constructed three different synthetic data sets, where each represents three different scenarios. Our approach is to first mathematically prove the behavior of the coefficients in each scenario and then visualize the proof with the help of an autocorrelation plot. The tool used for calculating the coefficients was written in C++ and is based on equation 3.1. The three different scenarios are discussed as follows:

- *Autocorrelation coefficients are near zero for a random data set* (white noise): To prove this argument we will make use of the equation 3.3. Since the data set is random, the elements separated by a lag of *k* will be uncorrelated. We can write the equation 3.3 as following:

$$r_k = \frac{\langle YY_k \rangle - m_y{}^2}{\langle Y^2 \rangle - m_y{}^2} = \frac{\langle Y \rangle \langle Y_k \rangle - m_y{}^2}{\langle Y^2 \rangle - m_y{}^2} = \frac{m_y m_y - m_y{}^2}{\langle Y^2 \rangle - m_y{}^2} = 0 \qquad (3.4)$$

  The above result suggests that for a random data set autocorrelation coefficients should be near zero for any and all time-lag separations. In order to confirm this proof, we generated a random data set using the *lrand48* C utility and then calculated the autocorrelation coefficients using our tool. The results of this experiment are shown in Figure 3-1.

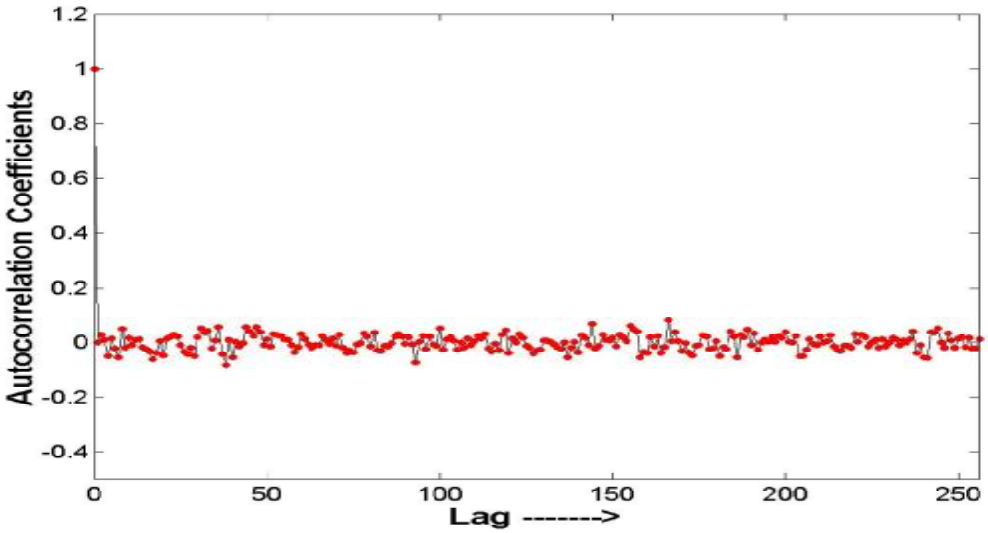


Figure 3-1: Autocorrelation plot for a random data set

- *One or more autocorrelation coefficients have a high absolute value for a data set that has explicit presence of repeating patterns*: To prove this argument, let us assume that data set Y_k, which is obtained by time shifting the original data set by a lag of *k*, is an exact replica of *Y*. We will further assume that the data sets are so large that they still have the same average value. For this scenario, we can write the equation 3.3 as follows:

$$r_k = \frac{\langle YY_k \rangle - m_y^{\,2}}{\langle Y^2 \rangle - m_y^{\,2}} = \frac{\langle YY \rangle - m_y^{\,2}}{\langle Y^2 \rangle - m_y^{\,2}} = 1 \qquad (3.5)$$

As shown, the above result confirms the autocorrelation coefficient for a repeating pattern will have an absolute non-zero value. For this scenario, we constructed a building block data set with a period (recurring distance) of 4, *i.e, Y* = 3, 7, 13, 19, 3, 7, 13, 19 … (length 1024). Figure 3-2 shows a spike at a lag of 4, followed by several spikes that are the harmonics of initial spike.

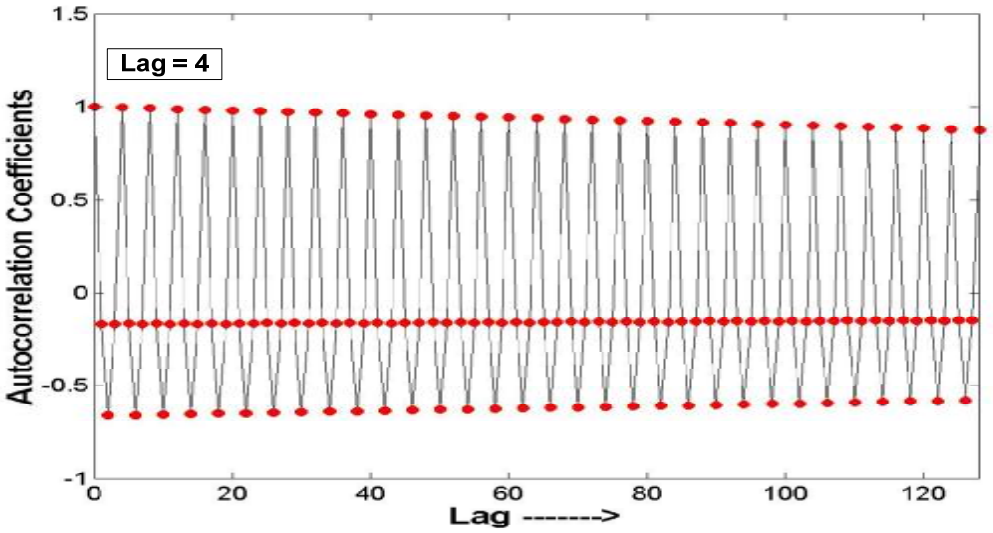


Figure 3-2: Autocorrelation plot for a data set with repeating pattern

- *Coefficients have a high absolute value even for the differential pattern hidden in the original data sets*: To prove this argument we will make of use of equation 3.2. We assume that the produced data set Y_k is equivalent to the data set $Y + x$, where x is a constant offset which we referred to as the ***stride***. Now we use equation 3.2 to prove the above argument:

$$r_k = \frac{\langle (Y - \langle Y \rangle)(Y_k - \langle Y_k \rangle) \rangle}{\langle (Y - \langle Y \rangle)^2 \rangle} = \frac{\langle (Y - m_y)((Y + x) - \langle Y + x \rangle) \rangle}{\langle (Y - m_y)^2 \rangle}$$

$$= \frac{\langle (Y - m_y)((Y + x) - \langle Y \rangle - x) \rangle}{\langle (Y - m_y)^2 \rangle} = \frac{\langle (Y - m_y)(Y - m_y) \rangle}{\langle (Y - m_y)^2 \rangle} = 1 \qquad (3.6)$$

The above computation proves that autocorrelation has the ability to detect even the differential pattern present in the data set. For this scenario, we constructed a data set of period of 3, *i.e*, Y = 2039, 4093, 8191, 2040, 4094, 8192, 2041, 4095, 8193 … (length 1024). Figure 3-3 shows spikes at lags that are multiple of *3* which confirms the above argument.

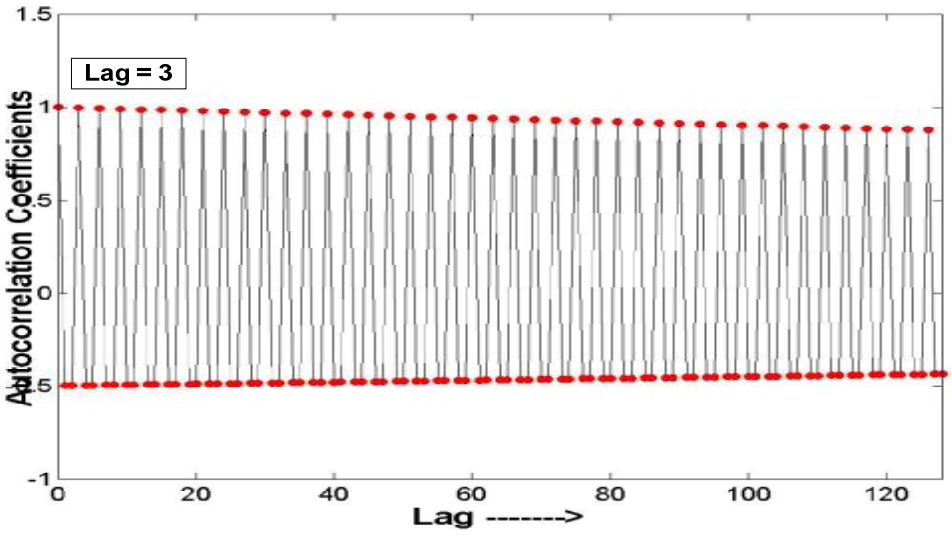


Figure 3-3: Autocorrelation plot for a data set with differential pattern

3.2 Autocorrelation Plots for Memory Intensive Benchmarks

In this section, we present the autocorrelation plots for our memory intensive benchmarks: *ammp, mcf, mgrid, mst* and *swim*. As we are interested in designing prefetchers that specifically prefetch from main memory, we generated plots only for the L2 miss stream of every benchmark. The miss streams were collected with the help of our simulator described in Chapter 2. The plots provide answers to the questions such as: Are there any patterns in the miss streams? What are the benchmarks that are marred with unordered addresses in their respective miss streams? etc. In addition to the characteristics of the pattern, autocorrelation also provides a rough estimate of the space requirements for detecting the pattern. For example, if the plot suggest that the elements of the pattern are recurring with a distance *x* than we need at least an *x* entry history table to faithfully detect the pattern.

While autocorrelation plots allow us to characterize the patterns, it fails in categorizing the patterns as absolute or differential. Answers to the question of the form – does the high absolute value of autocorrelation coefficient represent the absolute pattern or a hidden differential pattern? – are not available. To answer this question, we investigate both domains for the pattern suggested by the autocorrelation coefficients. We follow the methodology of first presenting the autocorrelation plots, and then present graphs reflecting patterns as recommended by the plots. Since the phenomenon we are analyzing depends on the properties of the workloads, it is reasonable to couple the workloads that reflect the similar kind of behavior into one sub-section. Thus, we first present pattern characterization of the workloads that have only ordered addresses in their reference stream and then we discuss the workloads that have both ordered and unordered addresses.

3.2.1 Applications with Ordered Addresses in their Reference Stream

We begin by presenting the set of autocorrelation plots for *ammp* and *mgrid*. These are the benchmarks that exhibit regularity in their miss stream and are highly predictable. Figure 3-4(a) shows the autocorrelation plot of the *ammp* benchmark. The plot contains isolated spikes and indicates predictable behavior in the stream. It can be seen that there is a spike at lag of 9589, followed by several smaller spikes that are harmonics of this initial spike. This implies that there exist a relationship between points separated by this many miss events. The results of the autocorrelation plot can be confirmed from Figure 3-4(b), where a snapshot of the miss stream of *ammp* benchmark is shown. There is a clear pattern that reappears by the same number of miss events.

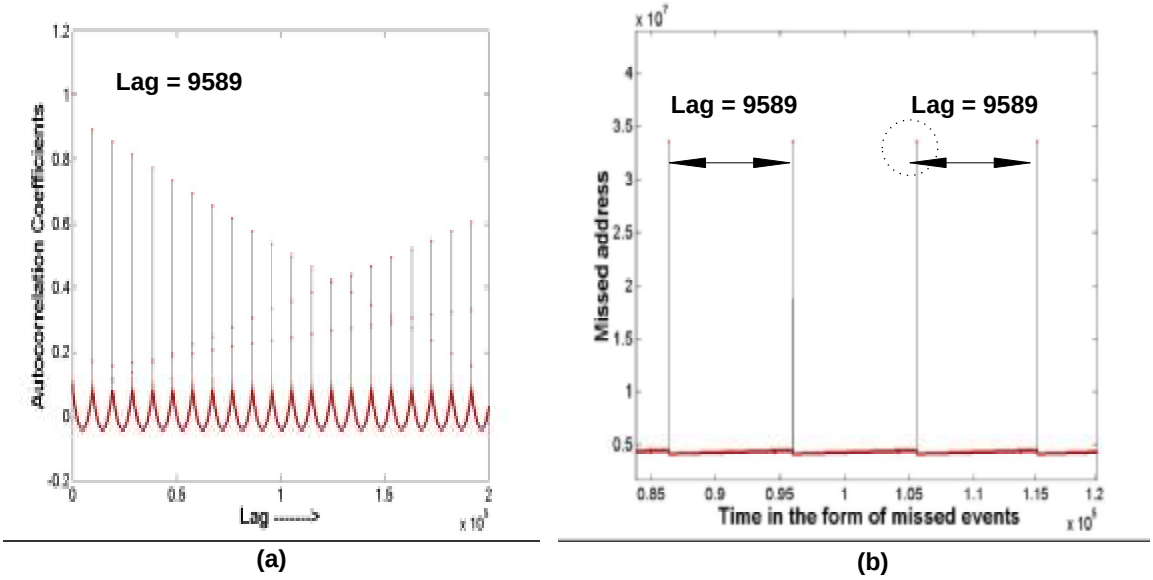


Figure 3-4: Autocorrelation plot and the underlying pattern of *ammp* benchmark

A closer look at Figure 3-4(b) shows that there exists a linearly increasing line in the repeating pattern. However, the autocorrelation plot does not show a high absolute value at lag of 1 (or 2, 3 etc.), corresponding to a constant stride in the differential domain. The reason for this contrary behavior is that one of the addresses (shown in dotted circle) is very large compared to other addresses and makes the variance significantly larger than the autocovariance for all the lags except the lag of 9589 (and multiples of 9589). Overall, we can conclude that there exist repeating patterns in the miss stream of *ammp* benchmark that reappear by the following recurring distances: *9589* in the absolute domain and *1* in the differential domain.

Similarly, an autocorrelation plot for *mgrid* benchmark is presented in Figure 3-5(a). In this plot, we see different levels of correlation – lag multiples of 6 in the range of 0.4 to 0.6 and lag multiples of 2 and 3 near 0.2 to 0.4 ranges. These results are counterintuitive since the actual pattern within the miss stream exists only at lags of 2 and 3. High correlation at a lag of 6 is an artifact of the constructive interaction of the

harmonics of lags 2 and 3. The existence of the patterns at 2 and 3 can be confirmed by looking at Figure 3-5(b) and(c). Although the absolute domain is being sampled these correlations correspond to the differential domain. It can be seen that there is no recurrence in the absolute domain, but it is obvious that the patterns exits in the differential domain where the elements of the pattern reappear by a recurring distance of 2 and 3, respectively.

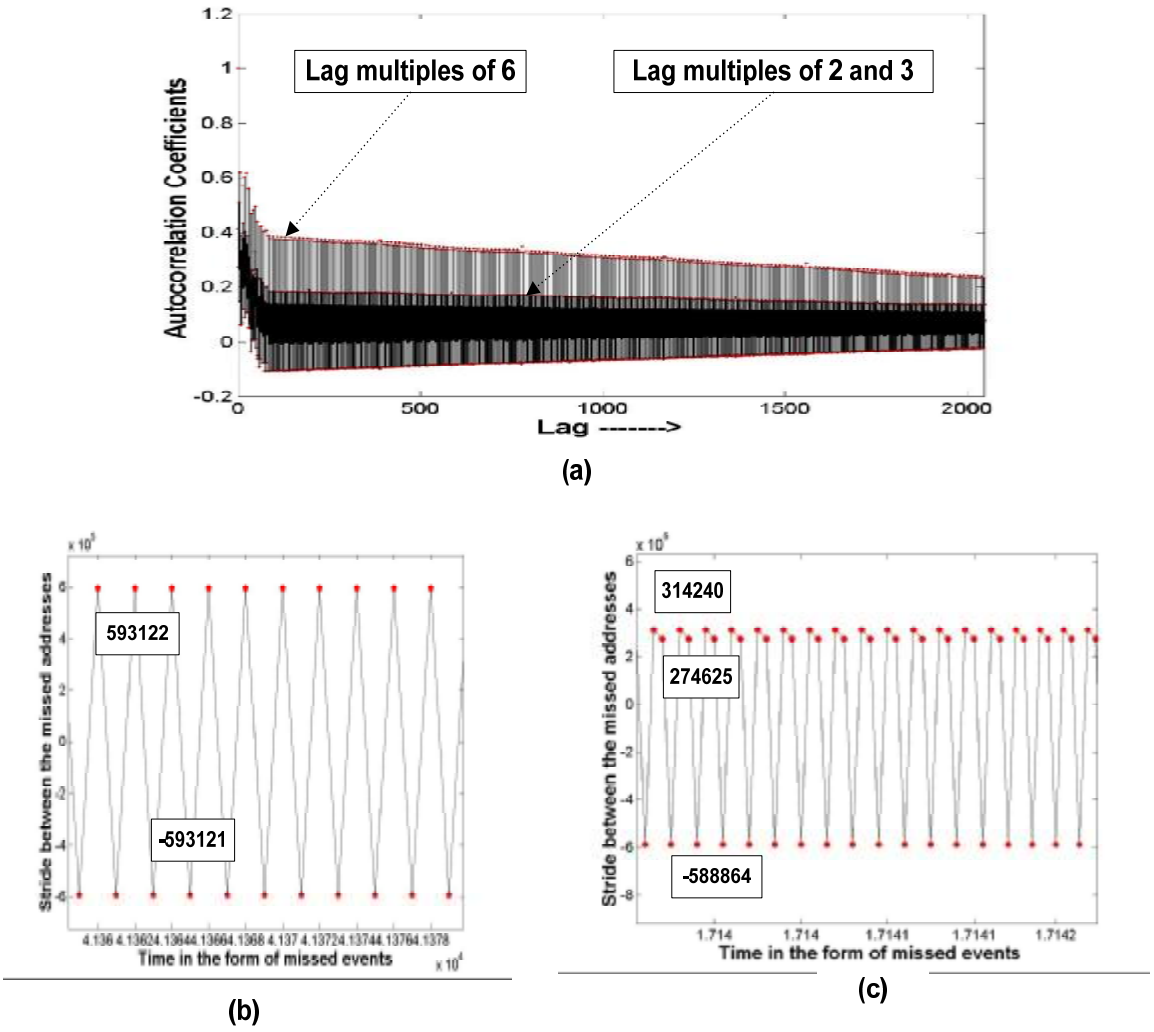


Figure 3-5: Autocorrelation plot and the underlying patterns of the *mgrid* benchmark

3.2.2 Applications with both – Ordered and Unordered Addresses

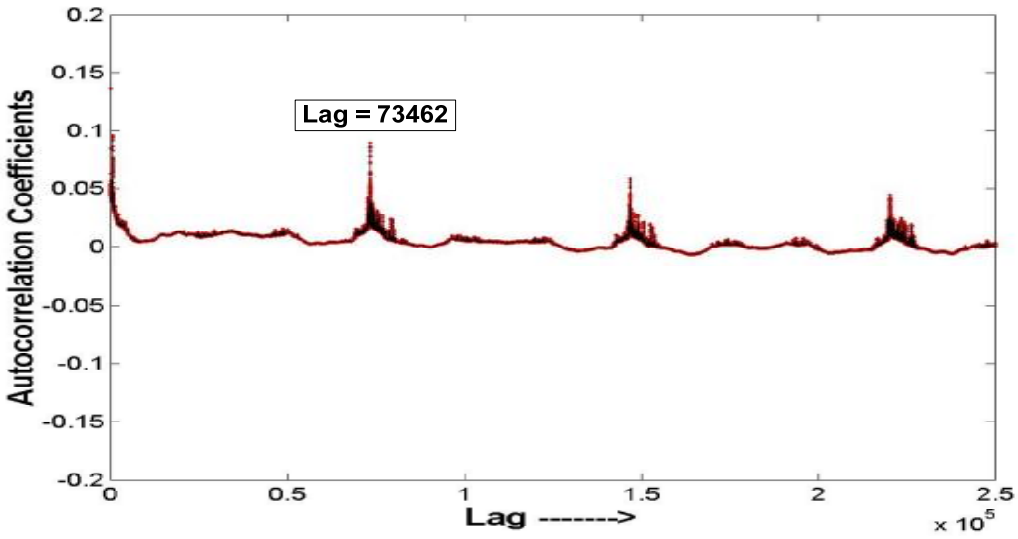


Figure 3-6: Autocorrelation plot for the *mcf* benchmark

Here we will discuss applications that have both ordered and unordered elements present in their reference streams. The first of this kind is *mcf* whose autocorrelation plot is shown in Figure 3-6. It can be seen that there is a spike in the plot around a lag of 73462, followed by several smaller spikes that are the harmonic of this initial spike. Indeed, the amplitude of this spike is small which corresponds to weak correlation, but this also implies that there exist some kind of relationship between points separated by this many events. There are two important implications of this plot. First, the spike in Figure 3-6 is not isolated – there is a ramp up to and from the lag of 73462 in the plot which conveys the presence of unordered addresses in the stream. The unordered addresses are primarily present due to the LDS transformation that happens all the time in the execution of *mcf*. Second, we more or less need a 73462 entry table to acquire the patterns present in this stream. The above implications can be confirmed from Figure 1-2 in Chapter 1 (for

brevity we do not reproduce the figure), where it was shown that the elements of the pattern reappear roughly with the ~73462 recurring distance in the absolute domain.

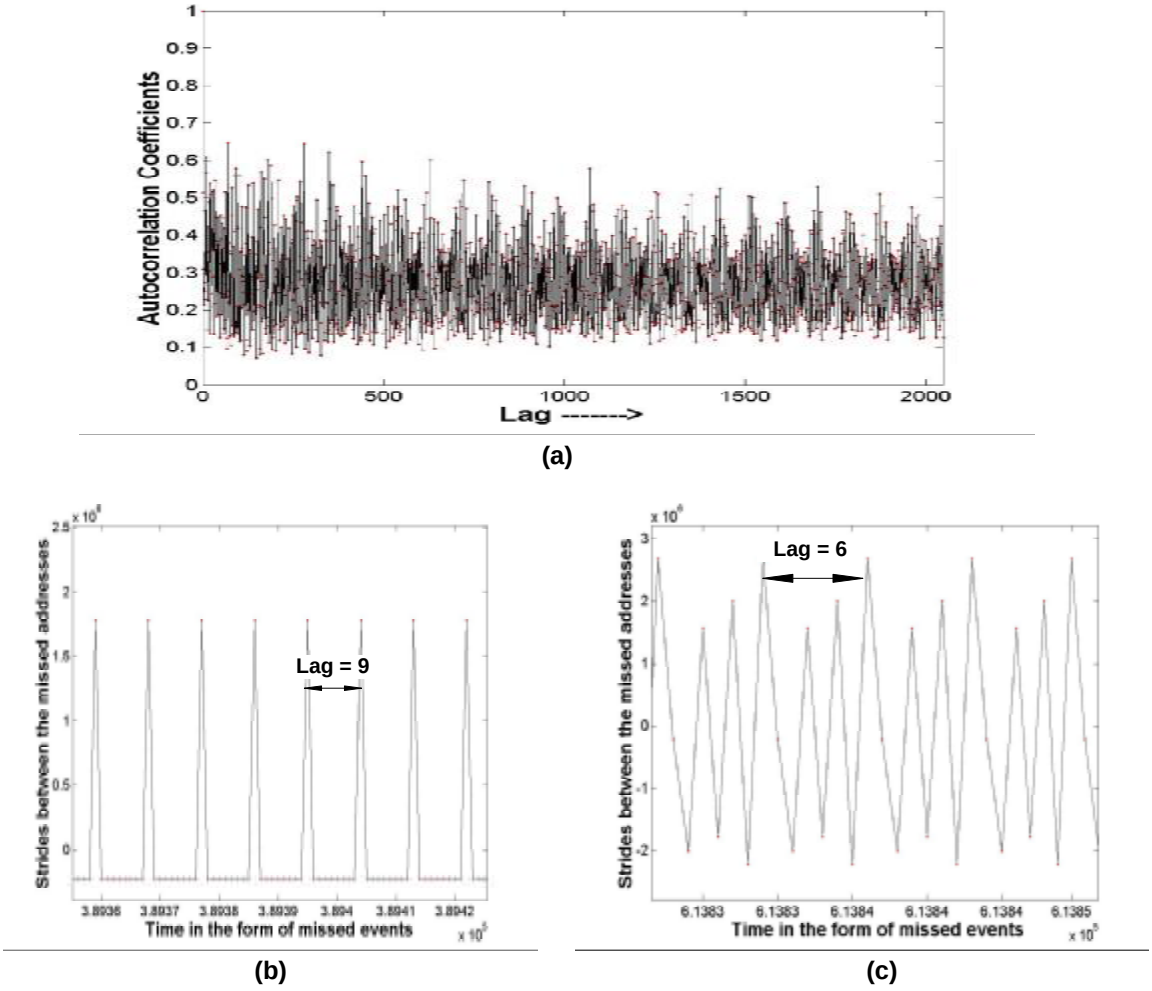


Figure 3-7: Autocorrelation plot and the underlying patterns for *swim* benchmark

The second application that shows both ordered and unordered elements in its reference stream is *swim*. Figure 3-7(a) shows the autocorrelation plot of the *swim* benchmark. As shown, the plot contains many sharp spikes and is definitely not a flat line. This conveys important information: first, there is a lack of isolated spikes that indicates the presence of unordered addresses and second, there are many patterns present in the reference stream of the swim benchmark. To check the authenticity of the autocorrelation coefficients, we looked into the miss stream and founded that the spikes

were caused by purely differential patterns and there was negligible recurrence in the absolute domain. Two of the major patterns that we observed in the differential domain are presented in the Figure 3-7(b) and (c). As can be seen, the elements of the pattern reappear by a recurring distance of 9 and 6, respectively. Furthermore, we also observed that some patterns were present in the company of unordered elements. An example of one such pattern is shown in Figure 3-8, where the presence of the unordered addresses changes the actual lag of the pattern from 10 to 20.

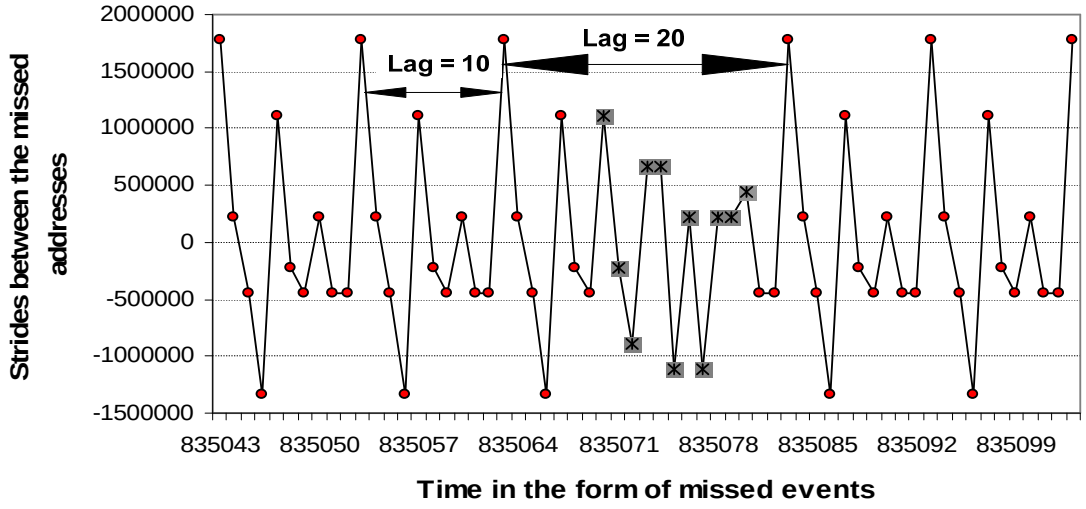
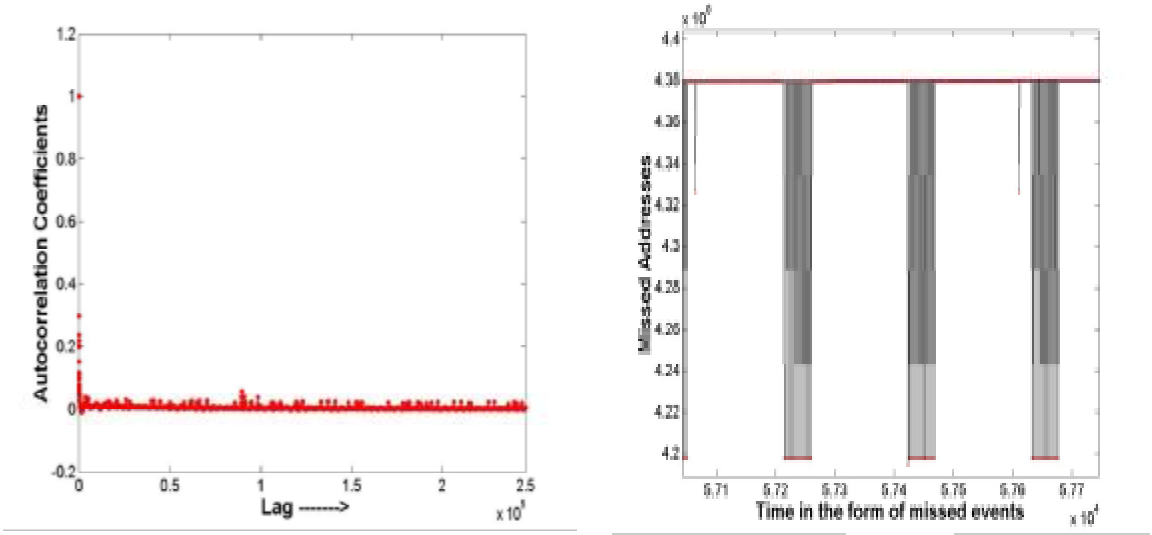


Figure 3-8: Pattern with a company of unordered addresses in the miss stream of *swim*

Although, *mcf* and *swim* illustrate patterns in their miss streams, there are applications like *mst* which exhibit a negligible amount of repeating patterns in both the absolute and differential domains. Figure 3-9(a) shows the autocorrelation plot of the *mst* benchmark. As shown, the plot contains no spikes or harmonics suggesting pattern and for most of the time lag separations coefficients are near to zero. There is a weak indication for the strided patterns since the plot shows few spikes near the lag of zero.

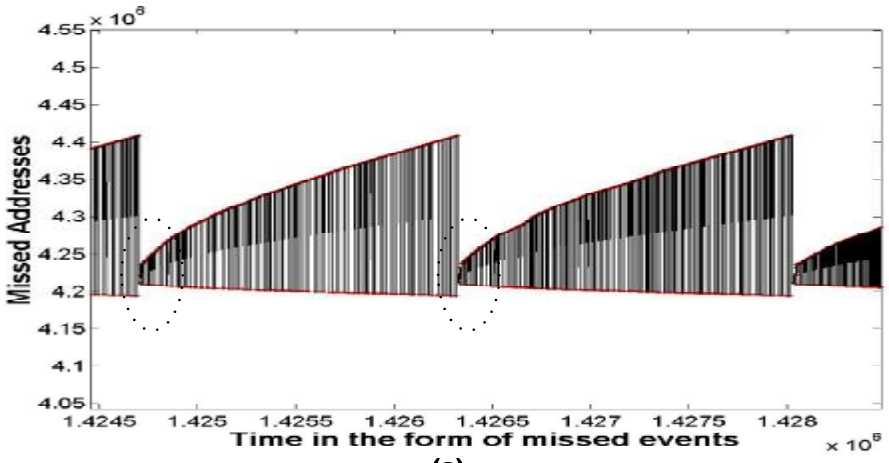
This can be confirmed from Figure 3-9(b), which shows the presence of a linearly increasing line in the miss stream.



(a) (b)

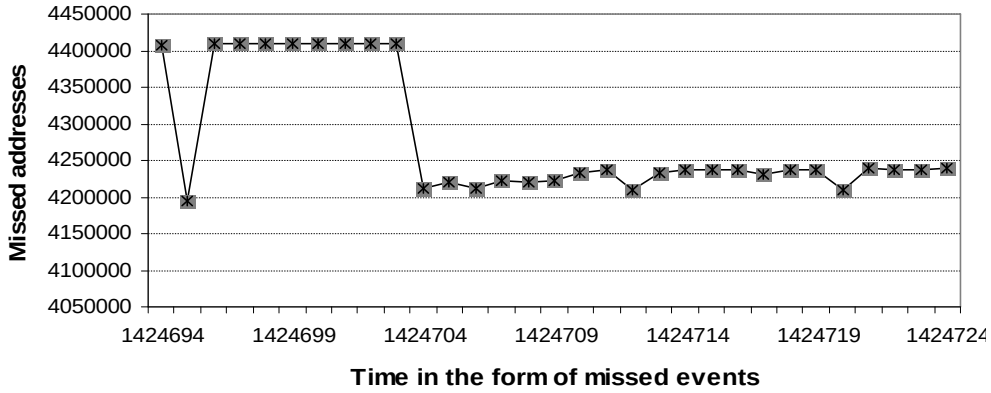
Figure 3-9: Autocorrelation plot and the underlying pattern for *mst* benchmark

However, the pattern shown in Figure 3-9(b) consists of less than 10% of the entire miss stream. The other 90% is comprised of the pattern shown in Figure 3-10(a). This pattern seems to be highly repetitive but is primarily composed of unordered addresses. The presence of unordered elements can be confirmed from Figure 3-10(b) and (c) where the snapshots of the original pattern are shown. These snapshots represent the misses enclosed by the dotted circle in Figure 3-10(a). As can be seen, no single datum reappears in the absolute or differential domain and thus confirms the insinuations of the autocorrelation plot. This suggest that the reference stream of *mst* is random and history based prefetchers will be ineffective in enhancing the performance of this benchmark. Moreover, care should be taken to design a prefetcher that avoids detecting these false patterns present among these set of benchmarks.



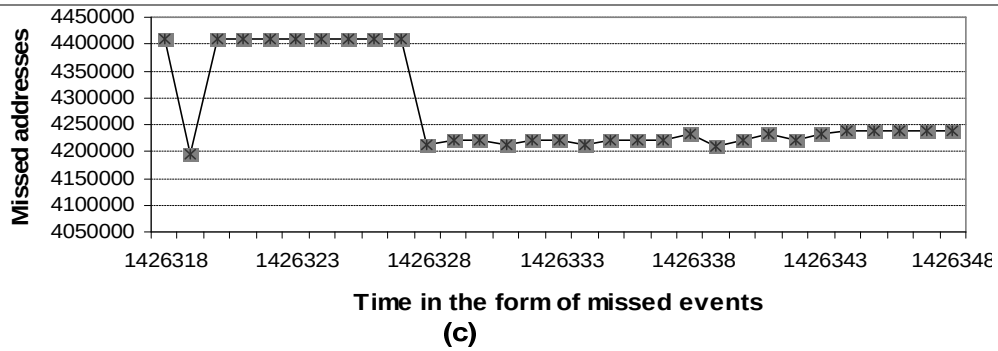
(a)

4408407, 4195004, 4408599, 4409079, 4408887, 4409015, 4409207, 4408823, 4408791, 4409366, 4211825, 4220832, 4212198, 4221983, 4220768, 4222175, 4231992, 4236597, 4209680, 4231960, 4236564, 4236565, 4237140, 4231896, 4236501, 4237748, 4209599, 4238483, 4237716, 4238099, 4238419



(b)

4408855, 4194988, 4409271, 4409463, 4409047, 4409239, 4409431, 4408983, 4409175, 4409367, 4212231, 4220865, 4222016, 4212199, 4220833, 4221984, 4212135, 4220769, 4221920, 4222208, 4231993, 4209729, 4222176, 4231961, 4222112, 4231897, 4237173, 4238132, 4237141, 4238100, 4237077



(c)

Figure 3-10: Pattern showing unordered addresses in *mst* benchmark

3.3 Autocorrelation Plots for Compute Intensive Benchmarks

In this section, we present the autocorrelation plots for our compute intensive benchmarks: *art*, *gcc*, *parser*, *twolf*, and *vpr*. Since these benchmarks exhibit negligible amount of cache misses for a 2 MB L2, we produced a miss stream for a 1 MB L2. The produced stream was then used for generating autocorrelation plots. The insights gained in this section will help us understand the properties of the compute intensive benchmarks.

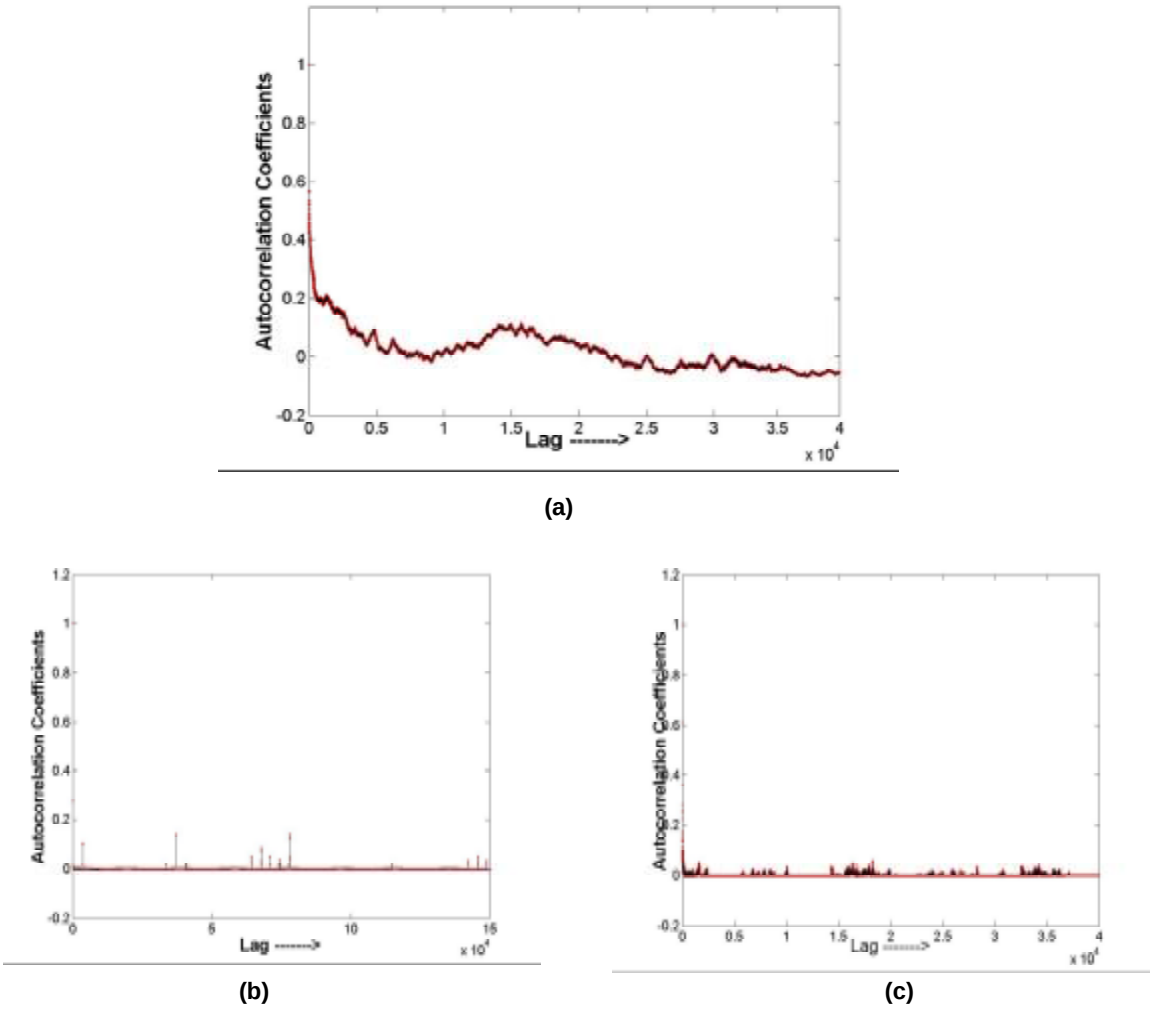


Figure 3-11: Autocorrelation plot for (a) gcc, (b) art, and (c) parser

Figure 3-11 shows the autocorrelation plots for *gcc, art,* and *parser* (Figure 3-11 (a), (b), and (c), respectively). There is a lack of isolated spikes that indicates the presence of unordered addresses in the reference stream of these benchmarks. For all these benchmarks, there is a strong indication for the strided patterns since the plots show spikes near the lag of zero. As a result, we can safely state that these benchmarks are predictable but have a smaller presence of unordered addresses.

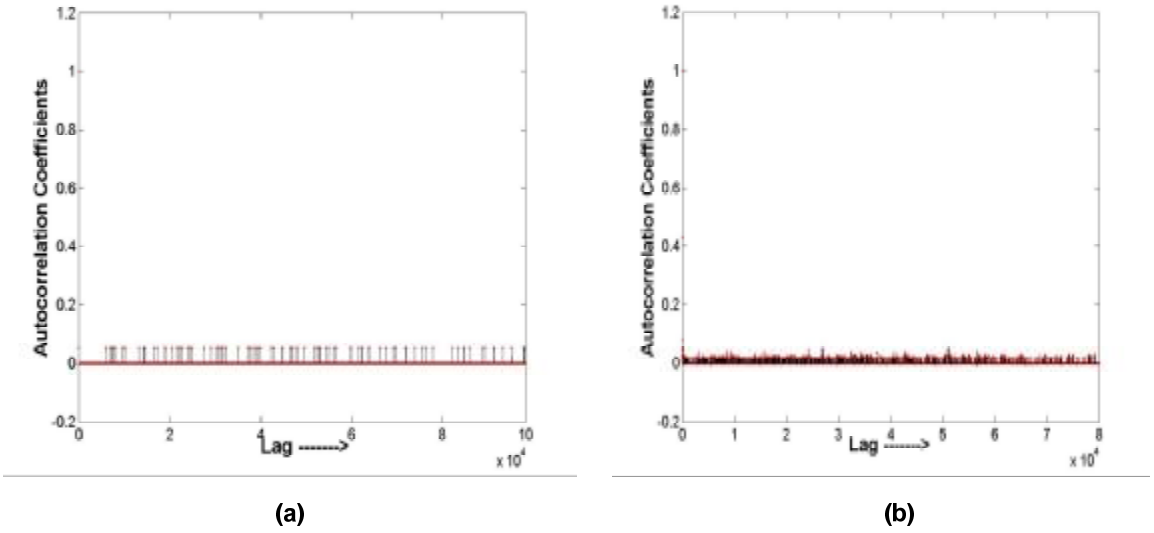


Figure 3-12: Autocorrelation plot for (a) twolf and (b) vpr

Figure 3-12 shows the autocorrelation plots for *twolf,* and *vpr* (Figure 3-12 (a), and (b), respectively). Similar to *mst, vpr* exhibit a negligible amount of repeating patterns in both the absolute and differential domains. Although autocorrelation plot of *twolf* exhibit spikes, the miss stream showed a very small percentage of repeating patterns. In our perspective, both of these benchmarks have significant amount of unordered addresses in their respective miss stream and thus are hard to predict.

3.4 Absolute vs. Differential Domain: A case study

In the previous sections we discussed and analyzed various miss characteristics of the different benchmarks. In this section, we make use of the insights that we gain from the autocorrelation experiments to make an argument for a prefetcher working in the differential domain. We refer to pattern of values as absolute patterns and pattern of stride between the values as differential pattern.

Detecting patterns in the differential domain is profitable in more ways than in the absolute domain. First, one differential correlation can represent many absolute value correlations and thus reduces storage requirements of the predictor. This can be observed from miss characteristics of *ammp* benchmark (as shown in Figure 3-4) where pattern reappears by a lag of 9589 in the absolute domain and by a lag of 1 in the differential domain. Second, a predictor working in the differential domain will further require less storage, since differences in values (strides) are generally smaller than the absolute values. Therefore stride values present in the differential domain acts like a crude form of compression for the absolute values. Third, the value stream that shows negligible recurrence in the absolute domain often exhibit hidden stride patterns in the differential domain. This is evident from the miss characteristics of benchmarks *mgrid* and *swim* (Figure 3-5 and Figure 3-7) where no patterns were observed in the absolute domain and high recurrence was exposed in the differential domain. Finally, not only the hidden differential patterns are revealed in the differential domain, but the original absolute patterns still exist in the differential domain.

However, differential domain has one shortcoming – the number of unordered elements increase in the differential domain. For example consider the address and stride stream below:

Addresses: A, B, C, D, R, A, B, C, D …

Strides: a, b, c, r', r'', a, b, c …

Here *A, B, C…* are the missed addresses and *a, b, c…* are the strides between the missed addresses. The example pattern is recurring with a lag of 5 in both the absolute and differential domain, and the unordered address *R* in the absolute domain gets mapped to *r'* and *r''* in the differential domain. Thus the increase in the number of unordered elements for the differential domain can be formulated as: $k * (n1 + n2 + n3 + \ldots) + k$, where k is the number of bursts of unordered addresses and n1, n2, n3 … are the sizes of the bursts, respectively. The implication of this phenomenon is that the address predictor working in the differential domain will have a lower accuracy than the predictor working in the absolute domain. Although differential domain has this problem, we will show in Chapter 5 that prefetcher using spectral prediction can be very effective in the differential domain.

3.5 Chapter Summary

In this chapter, we discussed and analyzed various miss characteristics of different benchmarks. We used probabilistic autocorrelation to detect and uncover the underlying patterns present in the reference stream. The autocorrelation results not only suggest the existence of periodic data misses in the reference stream but also show that unordered

addresses are very pervasive and are present in the access stream of most of the applications. The miss patters as suggested by the autocorrelation plots can be classified into three broad categories: highly predictable (*ammp,* and *mgrid*), predictable (*art*, *gcc*, *parser,* and *swim*), and hard to predict (*mcf, mst, twolf,* and *vpr*). Here, highly predictable corresponds to applications that exhibit regular patterns in their respective reference stream while predictable corresponds to applications that show a small presence of unordered addresses in their reference stream. The hard to predict are the ones whose reference stream has significant amount of unordered addresses.

Apart from discussing the individual pattern characteristics, the other contributions of the chapter include the following:

- We provided a mathematical background for analyzing the autocorrelation plots.
- We presented the advantages of the differential domain over the absolute domain.

Chapter 4 Spectral Prediction via Spectral Prefetcher

Spectral Prefetcher (SP) is our first proposed implementation of spectral prediction. The property that separates SP from other correlation-based prefetcher is its mechanism for detecting the patterns in the reference stream. A description of how this property is implemented in the design of SP covers the large portion of this chapter. SP, as proposed, divides the memory address space into Tag Concentration Zones (TCzones) and detects either the pattern of tags (higher order bits) or the pattern of strides (differences between the consecutive tags) within each TCzone. The prefetcher dynamically determines whether the pattern of tags or strides will increase the effectiveness of prefetching and switches accordingly.

We begin with a brief overview of SP where we identify its components and describe their working. Then in Section 4.2 we explain the operations of the prefetcher with the help of an example. Section 4.3 discusses the issues, such as “How to increase the timeliness of the prefetches generated by SP?” and “When and where SP fails in acquiring the patterns?” Here we also provide the solutions for these issues. In section 4.4, we do a sensitivity analysis to identify efficient configuration of various component

of the prefetcher. In Section 4.5 we experimentally evaluate SP. And, finally, we summarize this chapter and provide conclusions in Section 4.6.

4.1 Overview of the Spectral Prefetcher

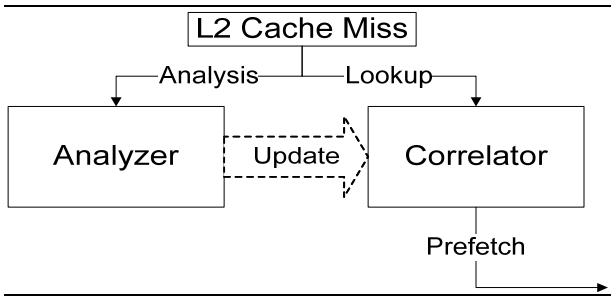


Figure 4-1: An abstract structure of the SP

In this section we will provide a brief overview of the Spectral Prefetcher (SP) and describe its components. Figure 4-1 depicts the abstract structure of SP. As shown, SP have two key components: *analyzer* and *correlator*. Analyzer encapsulates the *pattern history buffer* and *recurrence distance history buffer*, explained in Chapter 1, and detects pattern embedded in the miss stream. It also passes the pattern to the correlator which provides future predictions for prefetching.

4.1.1 Components of the Analyzer

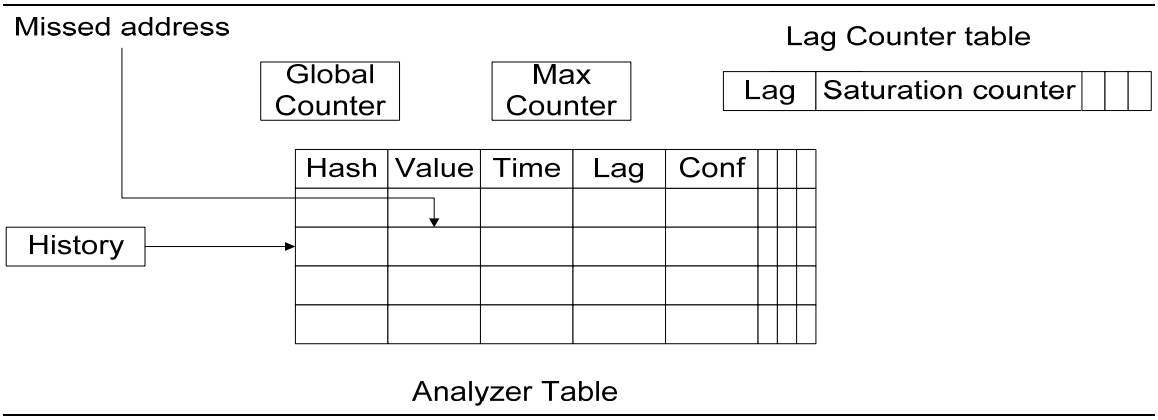


Figure 4-2: Structure of the analyzer

Figure 4-2 shows the structure of the analyzer. The function of the analyzer is to detect patterns by tracking the recurring distances of the missed addresses. To help with the rest of the chapter, we first discuss the components of the analyzer that are described as follows:

- *Analyzer Table*: This table is analogous to the *pattern history buffer* and maintains arrival records of the addresses observed in the miss stream. This is a set associative table whose index is generated by the previously observed address sequence. Each entry of the analyzer table has 5 fields. The first two fields store the correlated pair *i.e.* the hashed history and the missed address. The record of the recurring distances is kept in the *lag* field while the *time* field maintains the last time the corresponding pair was seen in the miss stream. The *conf* bit maintains whether the corresponding entry is a part of the pattern or not.

- *Global Counter*: Global Counter (GC) is analogous to the *position counter* described in Chapter 1. It is a continuous counter of miss events. During allocation of an entry in the analyzer table, the GC value is recorded in the time field. Whenever the reappearance of the correlated pair is detected, GC is used to calculate the recurring distance using the previously stored time value.

- *Max Counter*: Max Counter (MC) controls the sample size the analyzer can observe for pattern detection. When GC becomes equal to MC, analyzer stops the analysis and passes the pattern to the correlator.

- *Lag Counter Table*: Lag Counter Table (LCT) is analogous to *recurrence distance history buffer* and is used for maintaining information on how many correlated pairs

share the same recurring distance. LCT, as proposed, is a *k*-entry fully-associative cache-like structure in the analyzer. Each entry of LCT has two fields: *lag* and *saturation-counter*. The first field maintains a record of the recurring distance (or lag) by which the correlated pairs arrive in the miss stream. The second field maintains the number of pairs observed in the miss stream associated with that lag. If the saturation counter becomes equal to a given threshold value, the controlling logic assumes a pattern is detected.

The *history* register stores the last observed missed addresses in the stream.

4.1.2 Components of the Correlator

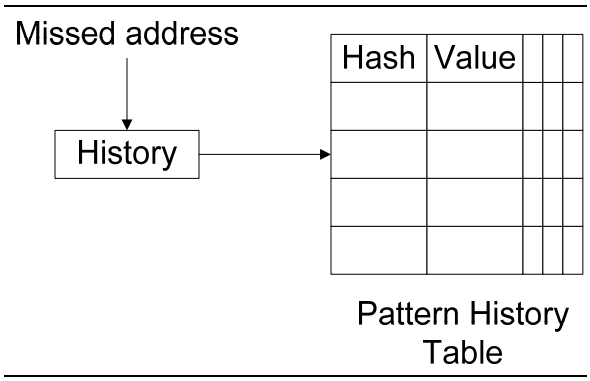


Figure 4-3: Structure of the correlator

Figure 4-3 shows the structure of the correlator. As shown, there are two important components: *Pattern History Table* (PHT) and a *history* register. PHT, as proposed, is a set associative table and stores the address correlation pairs for prediction. These correlated pairs are passed by the analyzer after examining them as part of the pattern. The index of PHT is generated by the previously observed address sequence present in the history register.

4.2 Operations of the Spectral Prefetcher

The operations of SP consist of three basic functions: *analysis, update* and l*ookup*. In the analysis phase, the analyzer tries to detect patterns by tracking the arrival records of the missed addresses, while in update phase it passes the pattern to the correlator. The lookup operation is performed by the correlator in order to predict a prefetch address based upon the knowledge of past address sequence in the miss stream. In order to understand these operations, we need to answer the following questions: i) how the reappearance of the correlated pair is detected, ii) what happens when reappearance is detected, iii) how does the sample size for analysis is controlled and iv) how and when the pattern is passed form the analyzer to the correlator. The first three questions cover the analysis operation while the last question covers the update operation. The answers to the above questions are as follows:

- *How the reappearance of the correlated pair is detected?* The correlated pair, referenced in this section, consists of the current missed address and the previously observed missed addresses in the history register. Reappearance of such a pair is detected while updating the analyzer table with the current missed address. First an index for locating the *analyzer table* set is generated using the truncated addition of the previously stored addresses in the *history* register. Second, from the analyzer table set, the entry tagged with the same hashed history is selected and its *value* field is matched with the current missed address. If both the value field and the current missed address match, we say reappearance has taken place.

- *What happens when reappearance is detected?* Once reappearance is detected, the recurring distance or the lag of the correlating pair is calculated as follows: *lag = GC – time field of the analyzer table entry*. Here *GC* or the global counter represents the number of miss events observed while the *time* field represents the previously recorded value of GC for the same pair. After calculating the recurring distance, an associative search of LCT is done to locate an entry with the same lag and its *saturation counter* is incremented.

- *How does the sample size for the analysis is controlled?* When a saturation counter for a lag entry in the LCT reaches a certain threshold value, the controlling logic assumes that elements of the pattern are reappearing with the same lag and changes the value of *max counter* (MC) as: *MC = current value of GC + saturated lag.* This dynamic change in the value of MC allows the pattern to be passed to the correlator sooner than a static, high value would. Once the MC has been dynamically updated, it is held static until the next analysis.

- *How and when the pattern is passed from the analyzer to the correlator?* When GC becomes equal to MC, the controlling logic stops analysis and broadcasts the saturating lags to the analyzer table. This in turn sets the *conf* field of the entries of the analysis table that have the saturating lags recorded in their respective *lag* field. Finally, an associative search is done across the analyzer table for locating the correlated pairs whose corresponding *conf* field is set and these pairs are passed to correlator for future predictions. At the end, the components of the analyzer, such as – GC, LCT and analyzer table are initialized and MC is set to a maximum value. This is done so the analyzer can once again start the operation of analysis.

A(0), B(1), ***R1***(2), C(3), D(4), E(5), F(6), ***R2***(7), A(8), B(9), C(10), D(11), E(12), F(13), ***R3***(14), A(15), B(16), C(17), D(18) ...

Figure 4-4: Example miss stream

With the basic understanding of the operations of SP, we turn to its mechanics with the help of an example shown in Figure 4-4. In this example the addresses are identified by letters. As shown, the pattern consists of elements: *A, B, C, D, E* and *F* that reappear by two different recurring distances: 7 and 8. The unordered addresses in the miss stream are represented as: *R1, R2* and *R3*. Let us assume, in the beginning, all the components of the analyzer are in the reset condition except MC, which is set to 255. For simplicity, it is further assumed that the threshold value for the saturation counter is 1 and the history register of the analyzer holds 1 previously seen missed address.

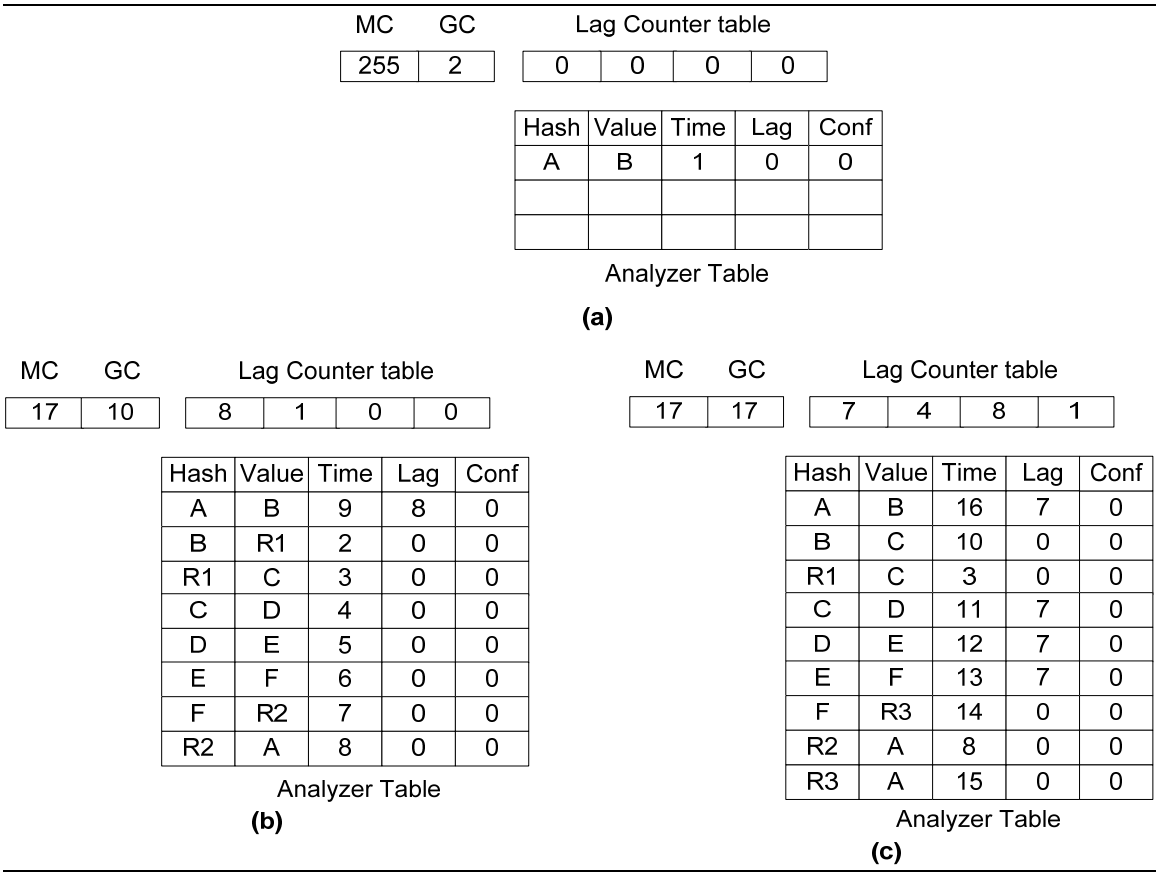


Figure 4-5: Example of analysis operation of SP

Figure 4-5 represents the steps taken in the analysis phase of SP. The first figure shows the condition of the analyzer after observing the first two missed addresses. Since there was no entry for the correlated pair of *AB* in the analyzer table, a new entry is allocated. The value of GC is passed to the time field, while the lag and conf fields are set to zero. The state of the analyzer after missed address B arrives for the second time is shown in Figure 4-5(b). When missed address B arrives for the second time, the values of history and GC are *A* and *9*, respectively. As there was an entry present in the analyzer table for the correlated pair of *AB*, the control logic detects reappearance and updates the lag field by the recurring distance, which is "GC – time" or 8. In addition, an entry is also allocated in the LCT to track the recurring distance observed in the miss stream. As the saturation counter in the LCT entry becomes equal to the threshold value of 1, the controlling logic assumes that the pattern is detected and updates the value of MC by "GC + saturated lag" or 17. The final stage of the analysis is shown in the Figure 4-5(c), where GC reaches the value of 17, and becomes equal to MC. In the meantime the analyzer detected the reappearance of pairs: *AB, CD, DE* and *EF*, which reappear at a distance of 7 in the miss stream.

The update state of SP is shown in Figure 4-6(a). In this operation, the saturating lags: 7 and 8 are broadcasted to the analyzer table, which in turn sets the *conf* fields of the corresponding entries. Finally, the entries whose conf bit has been set are passed to the correlator. As shown, the analyzer detects correlated pairs: *AB, CD, DE* and *EF* as part of the pattern and passes them to the PHT of the correlator for future predictions. Similarly, the example of lookup operation is shown in Figure 4-6(b), where the correlator calculates a prefetch address based upon the knowledge of the immediate past address

sequence. As shown, a missed address of *C* indexes into the PHT to generate a prefetch address of *D*.

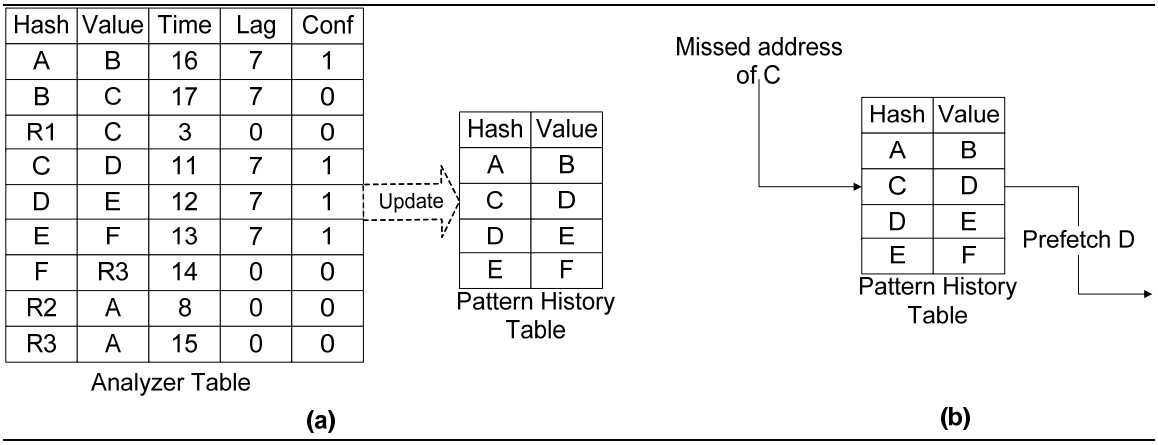


Figure 4-6: Example of update and lookup operation of SP

An apparent disadvantage of the update phase is the latency associated with searching the analyzer table for patterns. This latency, however, is mitigated by the fact that L2 caches miss takes hundreds of cycles to resolve, which can stall the processor, creating the window to work within. Having studied the structure and the operations of the spectral prefetcher, in the next section we present the individual solutions to the key issues that affect the performance of the spectral prefetcher.

4.3 Issues Involving the Design of the Spectral Prefetcher

In this section, we discuss issues regarding the implementation of the spectral prefetcher. The issues that need to be dealt with are: i) how to increase the timeliness of the prefetches generated by SP and ii) when and where SP fails in acquiring the patterns. In this section, we will provide solutions for the above stated issues and finally present a new design of SP incorporating these solutions.

4.3.1 How to Increase the Timeliness of the Spectral Prefetcher?

In order for a prefetching mechanism to be truly effective, not only it must have high coverage and accuracy, but it must provide prefetches in a timely manner. The prefetched data must be available in the cache prior to its use to completely mask the memory latency. Many prefetchers simply increase the prefetch degree, the number of prefetches triggered by single miss events to obtain timely prefetches [12, 18, 28, 29]. Unfortunately, this can have a negative impact on accuracy and complexity.

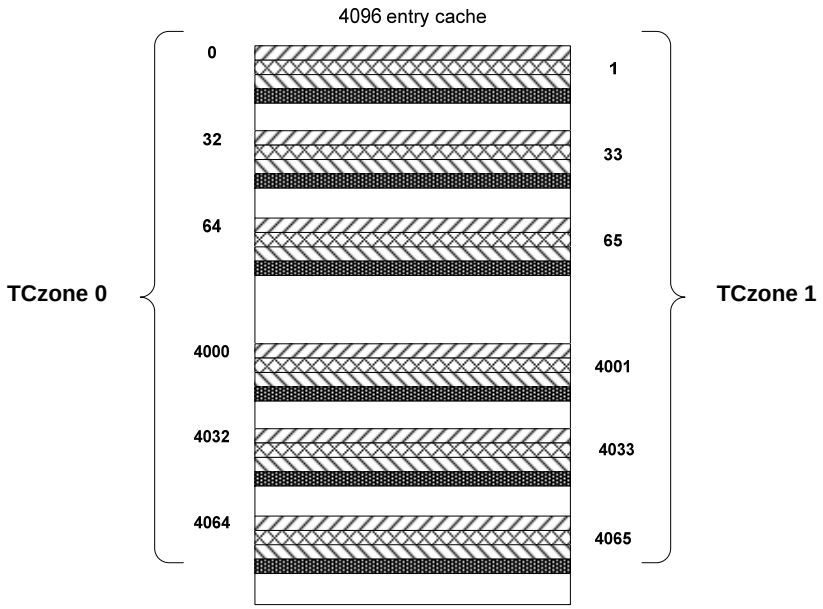


Figure 4-7: An example of 32 TCzone

For generating timely prefetches, we propose SP to correlate non-adjacent addresses in the miss stream. This is accomplished by partitioning the physical memory in Tag Concentration Zones (TCzones) and allowing SP to prefetch for these individual partitions. Two missed references are said to be within the same TCzone, if their addresses have the same lower order bits. Since memory is partitioned by the lower order bits, TCzones are strided across the memory, which in turn forces a fixed number of TCzones. Caches also partition the physical memory in similar fashion when viewed

from the perspective of the cache sets. In our simulation environment, the L2 cache has 4096 sets with 64-byte line size that naturally divides memory into 4096 TCzones, populated by 14-bit tags (assuming 32-bit machine). To divide memory into 32 TCzones from the previous setup, the seven most significant bits of the index are concatenated with the tag resulting in 21 bit tags (leaving 5 TCzone index access bits). An example of how 32 TCzone map into a 4096 set cache is shown in Figure 4-7.

Table 4-1: Average difference between the consecutive misses for different TCzones

Benchmarks	TCzone-1	TCzone-2	TCzone-4	TCzone-8	TCzone-16	TCzone-32	TCzone-64	TCzone-128
ammp	146	146	146	146	146	222	247	321
mcf	16	33	66	134	238	537	1075	2152
mgrid	305	610	1220	2441	4883	9766	19534	39068
mst	112	223	446	893	1803	3694	7355	14698
swim	148	297	593	1187	2375	4752	9504	19010

The prediction of tags within each TCzone produces timely prefetches because tag predictions span pages while applications usually access the same page several times before accessing another page. This is shown in Table 4-1, where the average time difference between the consecutive misses of the TCzones is shown for the memory intensive benchmarks. Our convention is to use TCzone-1 for the global miss stream, TCzone-2 when physical address space is divided into 2 separate partitions *etc*. As shown, for most of the benchmarks, the time difference between the consecutive misses increases exponentially with the number of TCzones. The exception is *ammp* whose memory accesses are non-uniformly distributed across the sets in a cache. This non-uniformity creates a heavy demand on some sets while other sets remain underutilized. As a result, partitioning the memory with TCzones has no effect in the time differences between the consecutive misses because all the misses are directed towards few TCzones.

4.3.2 A Shortcoming in the Current Design of the Spectral Prefetcher

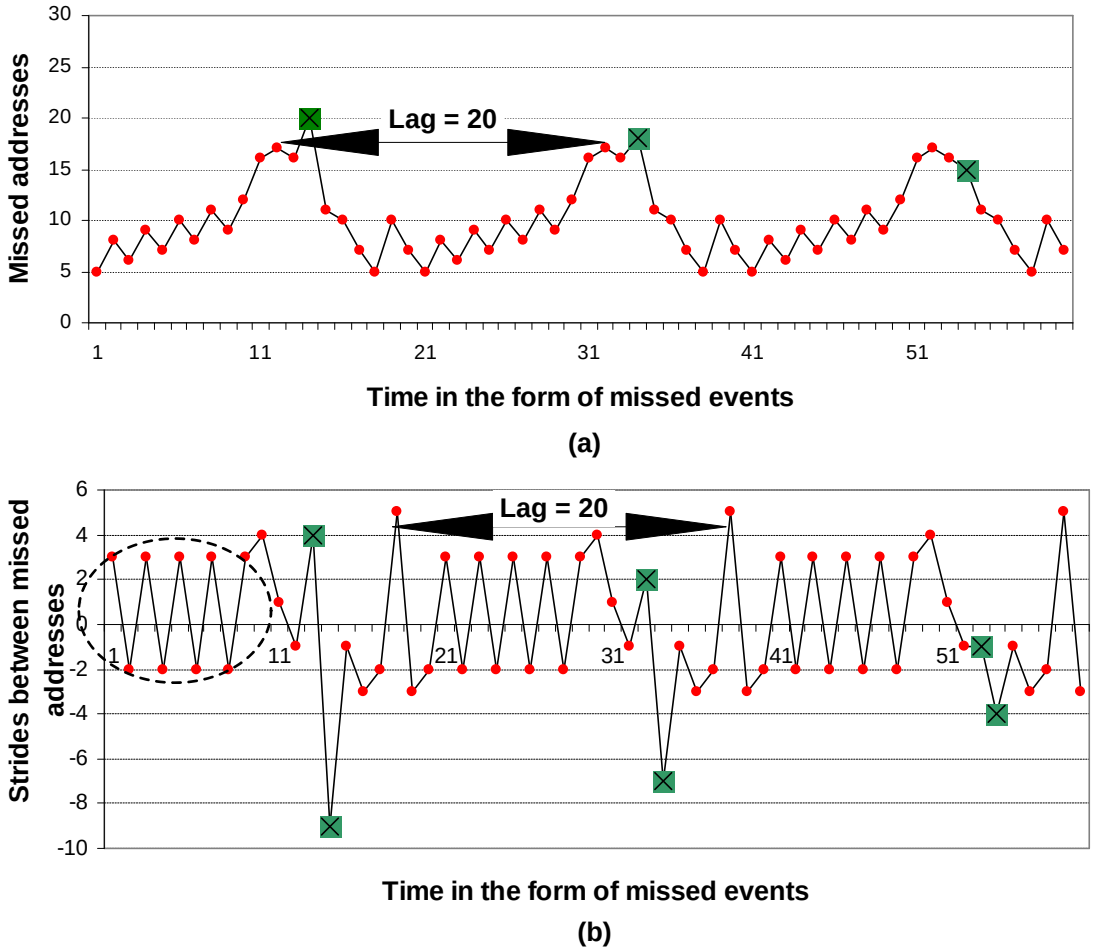


Figure 4-8: A synthetic miss stream

In the last section, we demonstrated the pattern detection ability of SP with the help of an example in the absolute domain. However, our current design of SP has a serious shortcoming in acquiring patterns in the differential domain. To illustrate this issue, we present in Figure 4-8 a synthetic miss stream where the top graph shows the pattern in the absolute domain and the bottom graph shows the same pattern in the differential domain. As shown, the triangle wave that is linearly increasing in the absolute domain translates into a repeating, high frequency pattern in the differential domain

(shown in dotted circle). This repeating high frequency component is detected by SP immediately, but the remaining larger portion of the pattern is never analyzed since the acquisition of the short, high frequency pattern initiates the update stage of the analyzer, which in turn resets the analyzer to start a fresh analysis. As a result, this phenomenon creates a distorted perception of the pattern and only part of the pattern is acquired by the prefetcher working in the differential domain. We term this phenomenon as: ***high frequency latching***.

However, high frequency latching leaves us with the dilemma. On the one hand, the results of autocorrelation plots in Chapter 3 favored differential domain over absolute domain, since it requires lesser space for detecting patterns in the reference stream. Moreover, it was also shown that the stream with negligible recurrence in the absolute domain often exhibit hidden stride patterns in the differential domain. On the other, the current design of SP fails in acquiring the pattern faithfully in the presence of high frequency components in the differential domain. This limitation is not the fundamental property of the spectral prediction rather it is an artifact of our particular implementation. To benefit from the differential domain while safe guarding against high frequency latching, we propose SP to be adaptive in nature – i.e., SP should alternate between absolute and differential domain whenever either is failing to acquire the pattern. This solution is not optimal, however, provides SP with means either to change from absolute to differential mode when there is no recurrence observed in the absolute domain or change from differential to absolute domain when high frequency components prevent the pattern acquisition in the differential domain.

4.3.3 A New Alternative Design of the Spectral Prefetcher

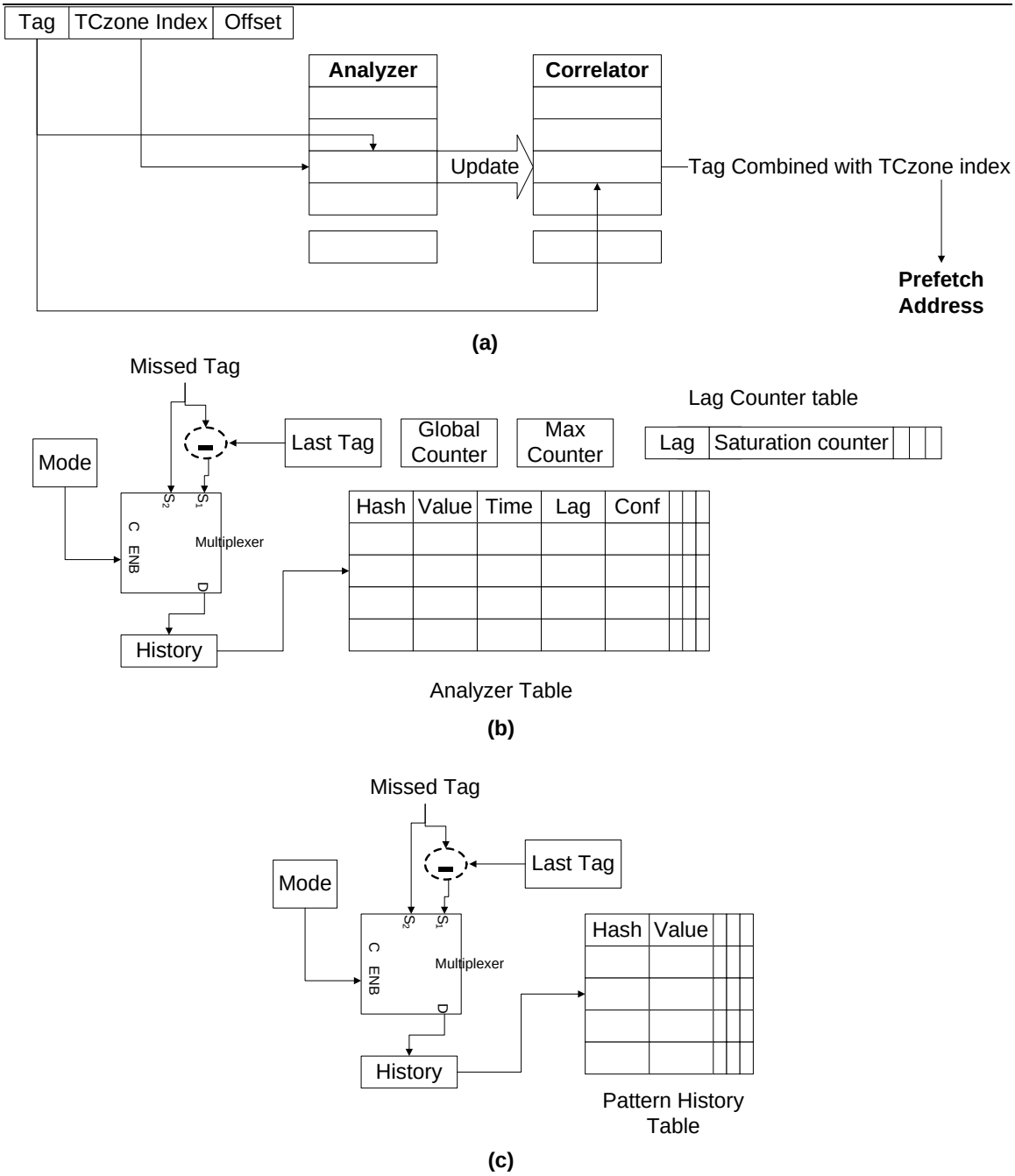


Figure 4-9: A new alternative design of the spectral prefetcher

In this section, we present a new design of SP that incorporates the solutions presented in the preceding sections. Figure 4-9(a) depicts the new structure of SP. As shown, this new form of SP partitions the memory into TCzones and detects tag pattern

within each zone. It allocates an analyzer and correlator for each zone, which are indexed by the index-access bits of the TCzone. It is further shown in Figure 4-9(a) that predicted tag is combined with the TCzone index-access bits to generate a complete cache line address. In order to attain a adaptive SP, two new components *last tag* and *mode* are added to the original design of analyzer and correlator, as shown in Figure 4-9(b) and (c). The function of the last tag is to maintain the last observed tag and is used in the differential mode to calculate the stride while mode bit tells the analyzer or the correlator whether they are in absolute or differential mode.

The adaptive property imposes few changes in the operations of SP, which are described as follows:

- *Analysis*: The scope of modification remains insignificant for this operation. The function remains similar to what described in the Section 4.2 – *i.e.*, analyzer in this phase detects patterns by tracking the arrival records of the missed tags. The only change is that analyzer now has the capability to detect either pattern of tags or pattern of strides between the tags. This is accomplished by transforming the history register to record either previously seen missed tags or the strides depending upon the mode bit. Similarly, the hash and the value field of the analyzer table are changed so that they can store either correlating pair of tags or strides. The form and functionality of the other components of the analyzer remains unaltered.

- *Update*: As previously explained, in the update stage, the analyzer passes the pattern to the correlator. In addition, we also need to pass the mode bit of the analyzer to inform the correlator that it is updated either by the pattern of tags or the pattern of

strides. The decision to switch between the absolute to differential mode (or vice versa) also happens in this state. If the control logic does not find any pattern *i.e.* no LCT entry matches the threshold value, control logic simply flips the mode bit of the analyzer.

- *Lookup*: In this operation, the correlator calculates the prefetch address based upon the knowledge of immediate past tag or stride sequence present in the history register. The value field of the PHT in absolute mode holds the next tag and in differential mode holds the next stride. Here, the entry tagged with the current history is selected from PHT and its value field is used to predict the next tag. If, in differential mode, the value field is added to the current missed tag to generate the next tag. Finally, the next tag is combined with the index-access bits of the TCzone to form the complete cache line address and, subsequently, a prefetch to this address is issued.

4.4 Sensitivity Analysis

In this section, we analyze in detail three aspects that affect the performance of SP: the selection of TCzone configuration, the size of the analyzer table, and the size of PHT. Sensitivity analysis of SP is important, since not all memory intensive benchmarks exhibit recurrence in both the domains. Moreover, the sizes of the patterns, either in absolute or differential domain, are not similar for all applications. We pursue the following methodology for these experiments:

- First, simulation results are presented for an infinite sized SP (32768 set 8-way associative analyzer table and PHT) using different TCzone configuration. To isolate

the impact of TCzone configurations, we present results for SP that monitors only in absolute or differential domain for detecting patterns. Here we select a TCzone configuration that performs reasonably well for all the benchmarks.

- Second, simulations results are presented where the size of the analyzer table is varied for an idealized TCzone configuration. For this experiment, we selected an infinite sized PHT to find out the optimal analyzer table size.

- Finally, simulation results are presented where the size of the PHT is varied for a specific sized analyzer table.

Table 4-2: Configuration of the components of SP

Components	Configuration of the components
Lag Counter Table (LCT)	An 8-entry fully associative LCT whose saturation counter saturates at threshold of 3
Max Counter (MC)	In the beginning of the analysis stage, MC is made equal to the number of entries the analyzer table can hold for analysis
History	This register stores two prior tags (or strides) in the miss stream

The configuration of other parameters of SP chosen for these experiments is shown in Table 4-2. These experiments were performed using the timing simulator described in Chapter 2. We extended the base simulator to incorporate SP for L2 cache prefetching, as described in Section 2.1.2. To keep the simulation requirements manageable, we only present the results for memory intensive benchmarks.

4.4.1 Impact of Varying TCzone Configuration

To show the impact of different TCzone configurations on the performance of SP, we present in Table 4-3 and Table 4-4 the coverage, accuracy, timeliness and performance gain results for memory intensive benchmarks. Here, we study eight different configurations of TCzones, *i.e.* partitioning memory into 1, 2, 4, 8, 16, 32, 64 and 128 zones. Table 4-3 and Table 4-4 show the impact of varying the TCzone configuration in absolute and differential domain, respectively.

For most benchmarks, we see that the performance increases as we increase the number of TCzones because partitioning memory into TCzones improves the timeliness of the issued prefetch requests. However, continual increase in the number of TCzones decreases the performance mainly because of two reasons. First, addresses directed towards a TCzone are less predictable than the addresses present in the global miss stream. This happens because the misses coming to a specific TCzone are widely apart in the global miss stream and are less consistent when compared to the adjacent misses of the stream. Second, prefetch requests generated for a specific TCzone can be too timely – *i.e.* the prefetch is issued so early that the data is discarded before it is used by the processor. The outcomes of the experiments illustrate both the positive and negative aspects of the TCzones. We make the following observations from these outcomes:

- First, we discuss the results of the *ammp* benchmark. The results demonstrate that partitioning memory into TCzones is not effective in increasing the timeliness of SP. In order to understand this contradictory behavior of *ammp*, we need to discuss its characteristics in more detail. Most of the missed cache line addresses of *ammp* follow a stride of 16 or 32 which makes the references non-uniform across the cache

structure. This creates a heavy demand on few cache sets while other sets remain underutilized. This is the reason why all the misses of ***ammp*** are targeted towards few TCzones. Consequently, partitioning memory into TCzones is not effective for generating timely prefetches.

- Second, we discuss the results of the *mcf* benchmark. The results suggest that *mcf* shows promising speedup in the absolute mode over differential mode. This is primarily due to the introduction of high frequency components in the differential domain. As a result, only a part of the *mcf's* pattern is acquired by SP and the remaining larger pattern is never analyzed in the differential domain. The results of SP functioning in the absolute domain imply that speedup for *mcf* increases as we increase the number of TCzones.

- Finally, we discuss the results of *mgrid, mst* and *swim.* As expected from the characteristics of these benchmarks, SP working in the differential domain boosts their performance. It is also evident from the results that timeliness of the prefetch request for these benchmarks increases with the number of TCzones. However, continual increase in the number of TCzones decreases the performance of *swim* because of the reasons discussed earlier in this section. Although, the resultant performance gain for *mst* is none, the results still shows that timeliness increases with number of TCzones. As previously mentioned in Chapter 3, the underlying miss patterns of *mst* are mainly comprised of unordered addresses and thus the coverage provided by SP is not commensurately high.

Table 4-3: Impact of varying TCzone configuration in absolute domain

Benchmarks	TCzone-1					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	48.17%	91.61%	23.62%	7.23%	69.15%	1
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-2					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	44.29%	91.73%	29.04%	6.03%	64.93%	1
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-4					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	41.57%	91.31%	42.95%	3.72%	53.34%	1.16
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-8					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	39.43%	91.37%	55.87%	2.55%	41.59%	1.16
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-16					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	38.27%	90.39%	71.55%	1.98%	26.48%	1.33
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-32					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	37.56%	89.96%	89.88%	1.02%	9.10%	1.41
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-64					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	34.06%	86.41%	93.45%	0.43%	6.12%	1.33
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Benchmarks	TCzone-128					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	96.12%	99.96%	50.08%	0%	49.92%	2
mcf	28.03%	82.15%	95.96%	0.24%	3.80%	1.16
mgrid	0%	0%	0%	0%	0%	1
mst	0%	0%	0%	0%	0%	1
swim	0%	0%	0%	0%	0%	1

Table 4-4: Impact of varying TCzone configuration in differential domain

Benchmarks	TCzone-1					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	15.36%	94.58%	46.45%	1.10	52.46%	1
mgrid	87.14%	99.94%	49.28%	6.18%	44.54%	1.61
mst	10.16%	81.19%	75.72%	3.65%	20.62%	1
swim	54.32%	96.72%	27.38%	6.43%	66.19%	1.15

Benchmarks	TCzone-2					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	13.07%	93.54%	55.25%	1.28%	43.48%	1
mgrid	86.83%	99.90%	58.70%	1.21%	40.09%	1.73
mst	10.80%	80.32%	79.57%	6.28%	14.15%	1
swim	48.74%	93.08%	54.61%	6.97%	38.42%	1.23

Benchmarks	TCzone-4					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	11.24%	95.01%	63.74%	0.81%	35.45%	1
mgrid	81.89%	99.78%	72.92%	0.80%	26.29%	1.87
mst	10.16%	79.97%	85.92%	6.26%	7.82%	1
swim	43.44%	93.34%	68.98%	2.65%	28.37%	1.38

Benchmarks	TCzone-8					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.88%	0%	49.92%	2
mcf	11.93%	95.23%	71.63%	0.73%	27.64%	1
mgrid	81.59%	99.78%	75.14%	0.99%	23.87%	1.87
mst	9.94%	77.81%	99.5%	0.17%	0.33%	1
swim	40.66%	96.92%	94.70%	1.28%	4.03%	1.38

Benchmarks	TCzone-16					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	10.33%	94.44%	78.37%	2.25%	19.39%	1
mgrid	79.19%	99.81%	82.15%	1.59%	16.26%	2.11
mst	8.61%	75.23%	99.83%	0.01%	0.16%	1
swim	37.65%	91.76%	98.28%	0.09%	1.63%	1.34

Benchmarks	TCzone-32					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	9.31%	94%	97.44%	0.56%	2%	1
mgrid	78.72%	99.56%	84.52%	0.04%	15.44%	2.13
mst	8.11%	75.44%	100%	0%	0%	1
swim	37.17%	91.76%	99.66%	0.08%	0.28%	1.34

Benchmarks	TCzone-64					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	8.17%	92.50%	99.91%	0%	0.09%	1
mgrid	74.72%	99.56%	98.68%	0%	1.32%	2.15
mst	8.79%	74.98%	100%	0%	0%	1
swim	18.13%	85.13%	99.97%	0%	0.03%	1.03

Benchmarks	TCzone-128					
	coverage	accuracy	timely	acceptable	poor	IPC gain
ammp	99.58%	99.88%	50.08%	0%	49.92%	2
mcf	7.60%	86.59%	100%	0%	0%	1
mgrid	73.28%	98.27%	99.91%	0%	0.10%	2.15
mst	8.54%	74.35%	100%	0%	0%	1
swim	14.88%	77.95%	100%	0%	0%	1

To summarize the results, we present the average gains across the memory intensive benchmarks for different TCzone configurations (Figure 4-10). As shown, dividing memory into 32 TCzones provides reasonable performance for memory intensive benchmarks in both the domains. For the rest of the chapter, we assume this TCzone configuration unless otherwise specified.

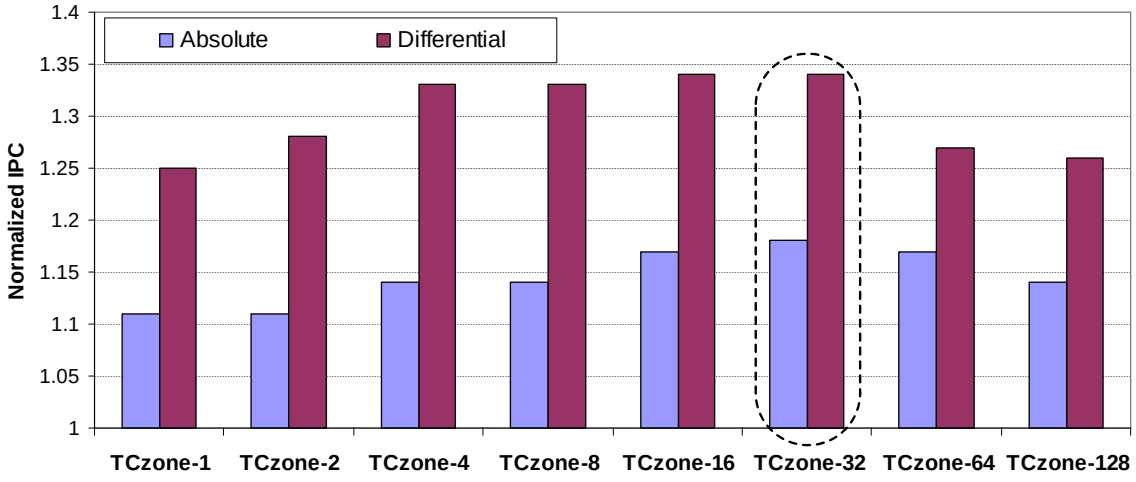


Figure 4-10: Impact of varying the TCzone configuration for SP

4.4.2 Impact of Varying the Table Sizes

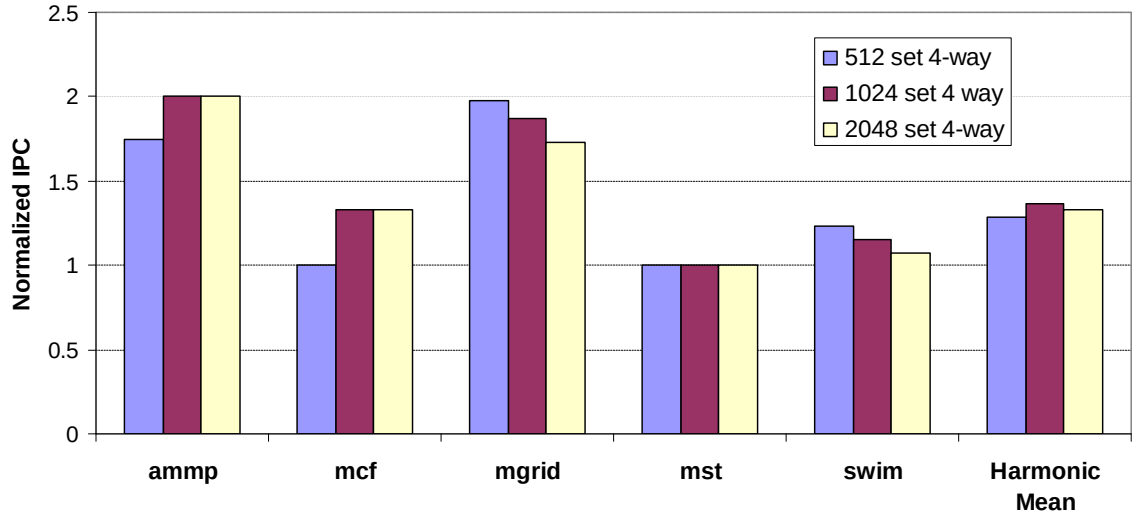


Figure 4-11: Impact of varying the analyzer table size

In this subsection, we varied the sizes of the analyzer and the pattern history table to find the optimal size for the general case. These tables, as proposed, are set associative and are accessed using an index value. As a result, if they are not large enough, the prefetcher will have difficulty in detecting and predicting the patterns. Figure 4-11 shows the impact of the analyzer table size (512 set 4-way to 2048 set 4-way) for an adaptive SP that partitions memory into 32 TCzones. For this experiment, we chose an infinite-sized pattern history table (32768 set 8-way).

As can be seen, *mgrid,* and *swim* show counterintuitive behavior and their performance decreases with the increasing size of the analyzer table. This occurs because SP favors the absolute mode initially and both of the applications show performance gain only in the differential mode. Moreover, the decision to transition from the absolute to differential mode particularly depends upon the size of the analyzer table, since the value of MC is initialized to the number of entries that the analyzer table can hold for analysis. As a result, there exists a switching delay, which increases with the size of the analyzer table, and is, thus, responsible for this unusual behavior. On the other hand, the results for *ammp* and *mcf* are straightforward where the performance increases with the size of the analyzer table. For a 512 set 4-way associative analyzer table, *mcf* shows negligible performance gains when compared to larger tables. The reason for this is smaller tables are not sufficiently big to contain the entire, large absolute pattern. Subsequently, the analyzer switches from absolute to differential mode more frequently, which further deteriorates the performance gain. It can be further observed that a 1024 set 4-way associative analyzer table performs reasonably well for all the memory intensive benchmarks. We select this configuration of the analyzer table for the rest of the chapter.

Similarly, Figure 4-12 shows the impact of varying PHT size (512 set 4-way to 2048 set 4-way) for a 1024 set 4-way analyzer table. It is worth noting that there is no increase in performance of the benchmarks for which SP function in the differential mode (*mgrid* and *swim*) because of the potentially smaller pattern sizes while performance of those for which SP function in absolute mode (*ammp* and *mcf*) increases to some extent. To satisfy a general solution that works well across all benchmarks, a 1024 set 4-way PHT was chosen as the optimal size.

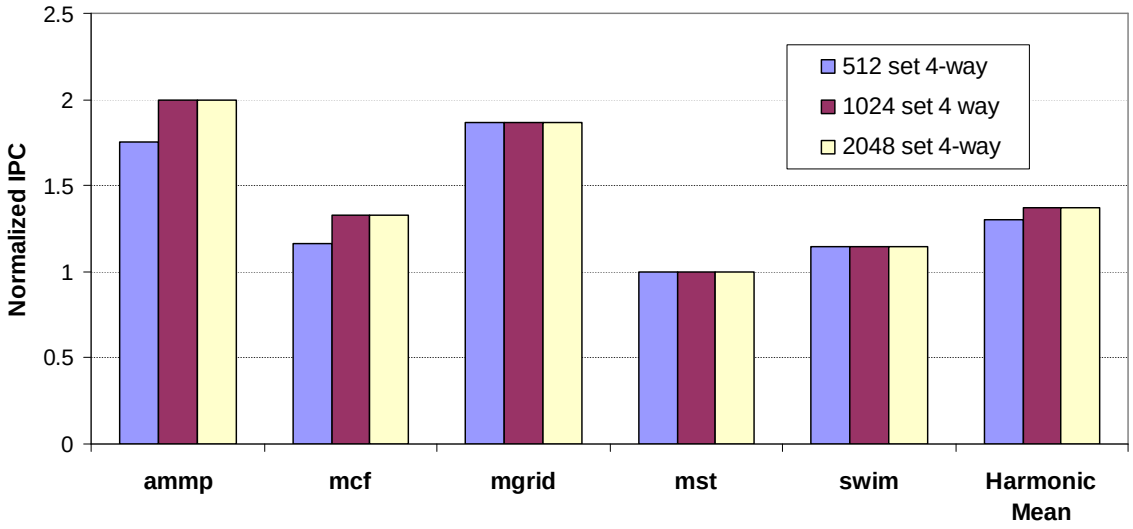


Figure 4-12: Impact of varying PHT size on the spectral prefetcher

4.5 Prefetcher Evaluation

In the prior section, we presented the individual solutions to the key issues that affect the performance of the spectral prefetcher. In this section, we combine these results into a single prefetching architecture and evaluate the effectiveness of SP in improving performance. We compare a 1 MB SP against a Tag Correlating Prefetcher (TCP) [15]

with a 2-MB correlation history table. Table 4-5 summarizes the prefetch table configurations for both the methods.

Table 4-5: Table configuration for TCP and SP

Prefetch Method	Table Configurations	Size
Tag Correlating Prefetcher (TCP)	4096 entry Tag History Table, 65536 set 8-way associative Pattern History Table	2 MB
Spectral Prefetcher	32 pair of analyzer and correlator with both analyzer and PHT as 1024 set 4-way associative, 8-entry LCT whose saturation counter saturates at threshold of 3	1.1 MB

TCP is selected for comparison as it also divides memory in a strided fashion (TCzones) and predicts pattern of tags for a given cache set. As previously discussed, our simulation environment uses a 2 MB L2 with 4096 sets, thus making TCP divide memory into 4096 TCzones. The indexing scheme that we used for the correlating table of TCP is similar to SP, where truncated addition of two previously stored miss tags is performed. In addition, the hash generated for indexing is combined with the miss index of the cache so that each cache set has its own private space in the correlating table. In order to gauge the best performance of TCP, we assume a latency free environment.

In accordance with the results presented in Section 4, we use an adaptive SP. The configuration of the analyzer table and the PHT are selected as 1024 set 4-way associative. For performance evaluation, we consider a practical implementation of SP, where address prediction by the correlator takes seven cycles. In addition, the latency associated for the analysis of miss tag by the analyzer was also set to seven cycles. The latency associated for passing a correlated pair (hash history and the corresponding next tag or stride) was chosen to be four cycles. Thus, passing 10 correlated pairs from

analyzer to the correlator will require 40 cycles and no analysis or prediction will take place during this period. The size of the analyzer and correlator can be calculated using the following formulae:

$$Size\ of\ \ analyzer = number\ of\ \ entries\ in\ the\ analyzer\ table * sizeof(hash + value + lag + time) + number\ of\ \ entries\ in\ LCT * (lag + saturation\ counter) + size\ of\ \ GC + size\ of\ \ MC$$

$$Size\ of\ \ correlator = number\ of\ \ entries\ in\ PHT * sizeof(hash + value)$$

In our simulation environment, we selected the following sizes for the fields of analyzer table and PHT entries: 5-bit hash, 19-bit value, 11-bit lag, and 11-bit time, respectively. In addition, the sizes of both GC and MC were selected as 12 bits, making the size of the analyzer and correlator ~23.056 and ~12 KB, respectively. Since there are 32 pairs of analyzer and correlator corresponding to 32 TCzones, the total size of SP used in the simulation became 1.1 MB.

Table 4-6: Coverage and accuracy results for TCP and SP

Benchmarks	TCP		SP	
	coverage	accuracy	coverage	accuracy
ammp	98.48%	99.17%	96.12%	99.16%
mcf	39.45%	41.35%	31.26%	85.73%
mgrid	0%	0%	64.28%	97.27%
mst	13.76%	19.35%	8.11%	75.43%
swim	0%	0%	37.13%	91.78%

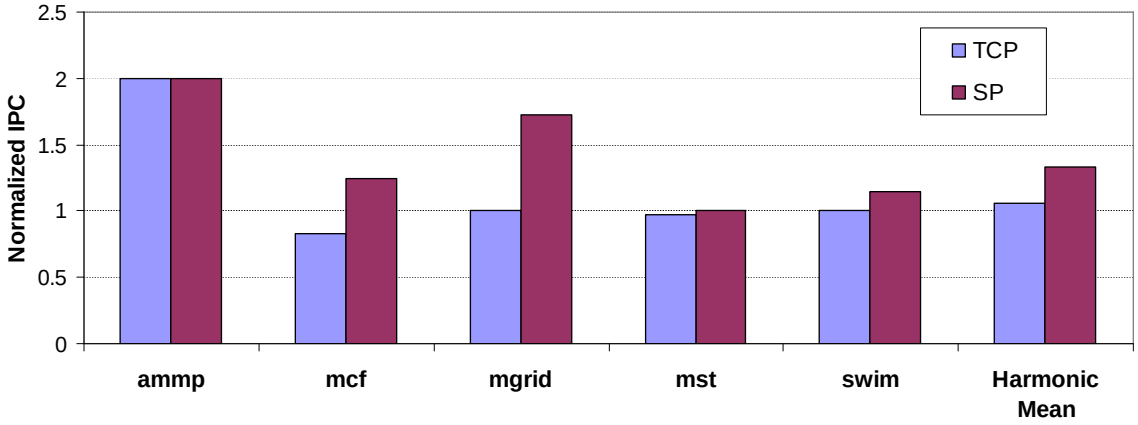


Figure 4-13: Performance comparison of SP with TCP

Figure 4-13 compares the performance results of SP against TCP with a 2MB correlation table. Table 4-6 shows the coverage and accuracy results of both the prefetching mechanisms for memory-intensive benchmarks. SP outperforms TCP on all memory intensive benchmarks, except *ammp*. On average, SP achieves a 1.33 performance improvement while TCP shows a performance improvement of 1.05. We observe that TCP shows performance gain only for *ammp* benchmark because this application exhibit regularity in absolute domain and is highly predictable. For benchmarks that favor differential domain (*mgrid* and *swim*), SP outperforms TCP by a huge margin. This happens because TCP can only acquire patterns in the absolute domain where these applications exhibit minimal repeatability. Moreover, TCP follows strict value locality by recording every instance of misses *i.e.* both the patterns and unordered addresses for prediction, and thus results in low prediction accuracy. This is evident in the result for *mcf* and *mst*, where TCP actually incurs performance degradation. On the other hand, SP has a unique ability to distinguish between the repeating patterns and the unordered addresses, which makes the predictions of SP highly accurate and increases performance.

4.6 Chapter Summary

In this chapter, we introduced and studied the first implementation of spectral prediction: *Spectral Prefetcher* (SP). We presented the basic design concept and explained the operations of SP with the help of an example. We then proposed SP to divide the memory into Tag Concentration Zones (TCzones) for generating timely prefetches. This new form of SP predicts pattern of tags for each individual zone by allocating a pair of the analyzer and the correlator for each zone. We also discussed a potential weakness of SP in acquiring the patterns effectively in the presence of high frequency elements. This weakness motivated us to design SP to be adaptive in nature – *i.e.*, SP alternates between absolute and differential domain whenever either is failing to acquire the pattern. Finally, we have shown that spectral prefetcher performs well, achieving high prediction accuracy on the memory intensive benchmarks.

As previously noted in the Introduction, SP uses a rather complicated algorithm for implementing spectral prediction. Nevertheless, we must ask ourselves whether there are simpler ideas that implements spectral prediction without increasing the complexity that the microarchitect has to deal with. In the next chapter, we explore an alternative design that provides better performance improvements while employing a simplified algorithm for implementing spectral prediction.

Chapter 5 Spectral Prediction via Differential-only Spectral Prefetcher

As indicated by the previous chapter, the Spectral Prefetcher (SP) employs a complicated algorithm for implementing spectral prediction. In this chapter, a more elegant implementation of spectral prediction is proposed: *Differential-only Spectral Prefetcher*. The proposed prefetching mechanism evolved through the continuous simplification of the SP algorithm. In this process, we identified answers to the questions such as, "What are the limitations in the design of SP?" and "How best can we overcome these limitations?" However, SP was the first of its kind to implement a spectral technique for speculation, and hence, the understanding required for coming up with an efficient design was not available. Only after designing SP, different possibilities became apparent for exploiting spectral prediction.

Thus, in this chapter, we begin by discussing the limitations of SP and their individual solutions. We then put the individual solutions together into a single prefetching architecture – Differential-only Spectral Prefetcher (DOSP) – which provides better performance using significantly smaller hardware structures. This new prefetching architecture detects only stride patterns within the global miss stream and has the unique property of tying two non-adjacent strides for generating timely prefetches. Apart from

presenting a new simplified prefetcher, the other contributions of this chapter include the following: 1) we describe a hardware implementation for correlating two non-adjacent values in the stream; 2) we present a simulation driven empirical evaluation of DOSP in a uniprocessor environment and 3) we also evaluate the performance benefit of DOSP using a set of co-scheduled pair of benchmarks on a dual core CMP.

5.1 Limitations of the Spectral Prefetcher

In this section we are concerned with how to use spectral prediction to our advantage – *i.e.* how to increase the performance potential of its first implementation: Spectral Prefetcher (SP). To find the answer of the above question, we identified four limitations in the original design of SP. Here limitations are the issues that either hamper the effectiveness of SP or increase its power consumption. These limitations and their individual solution are as following:

- *Discontinuity in analysis of pattern*: SP employs a "discrete window" for pattern acquisition. At the end of the detection stage, patterns are passed from the detection unit (analyzer) to the prediction unit (correlator) and subsequently the detection unit is initialized to start a fresh analysis. This heuristic works only when the pattern is completely acquired by the detection unit. However, if only a subcomponent of the pattern is detected (presence of high frequency components *etc.*), the remaining portion of the pattern will be lost due to the initialization process. This scenario creates a distorted perception of a pattern and only a part of the pattern is acquired. The solution is to use a "continuous window" for pattern detection. For accomplishing this we propose two changes in the design. First, the resetting of the

detection unit should be avoided. This in turn avoids the loss of useful information and provides continual analysis of the stream. Second, the position count (GC) should be made as the modulus count of the miss events. This has to be done because the entries of the detection unit can become stale upon the overflow of the position counter.

- *Expensive operations*: SP uses associative search across the detection unit for passing information to the prediction unit. Moreover, passing of patterns may require multiple ports in the prediction unit. We would like to avoid these costly operations so as to make the prefetcher implementable. Our solution is to use a single structure for detecting as well as predicting the pattern. This solution eliminates the need for associative searches, since no transfer will be required. In addition, this solution will also reduce the presence of *stale data* in the table because correlated pairs that are no longer a part of the pattern will be quickly recognized and no predictions will be made using them.

- *Significant investment in hardware*: SP needs a 1 MB hardware budget for effectively acquiring patterns. This size requirement is comparable to current on-chip L2 caches and therefore brings up concerns about latency and power overhead. We would like to keep this structure small so as to make it hardware realizable. Yet we would also want to achieve high performance gains. We can cater to both requirements by having a small prefetcher functioning only in the differential domain. Recall that SP required structure sizing for the worst case needs of the absolute domain because it is marred with high frequency latching in the differential domain. Since the problem of high frequency latching disappears using continual analysis of the pattern, we can design a

mechanism that works only in the differential domain and can exploit the reduced size benefits that differential have to offer.

- *Partitioning memory into TCzones adds complexity*: SP partitioned the memory into TCzones for generating timely prefetches. It also allocates a pair of pattern detection and prediction unit for these partitions. This approach adds complexity to already time-sensitive structures (analyzer and correlator) and may require several ports for effectively detecting and predicting patterns across this array of structures. Moreover, TCzones are not effective for applications that access only few cache sets and primarily exhibit conflict misses. An example of such an application, ***ammp***, was discussed in the last chapter. We propose a second option: ***correlation queue***, a mechanism that facilitates in correlating non-adjacent addresses in the global miss stream for generating timely prefetches.

The recommended changes laid a foundation for a new form of the spectral prefetcher, which not only uses dramatically less size but also provides improved performance. The next section describes our second implementation of the spectral prediction.

5.2 Differential-only Spectral Prefetcher

In this section, we will discuss the mechanics of the Differential-only Spectral Prefetcher (DOSP). This section provides the brief introduction to the DOSP architecture, describing its basic operations and introducing the correlation queue for generating timely prefetches.

5.2.1 Components of the Prefetcher

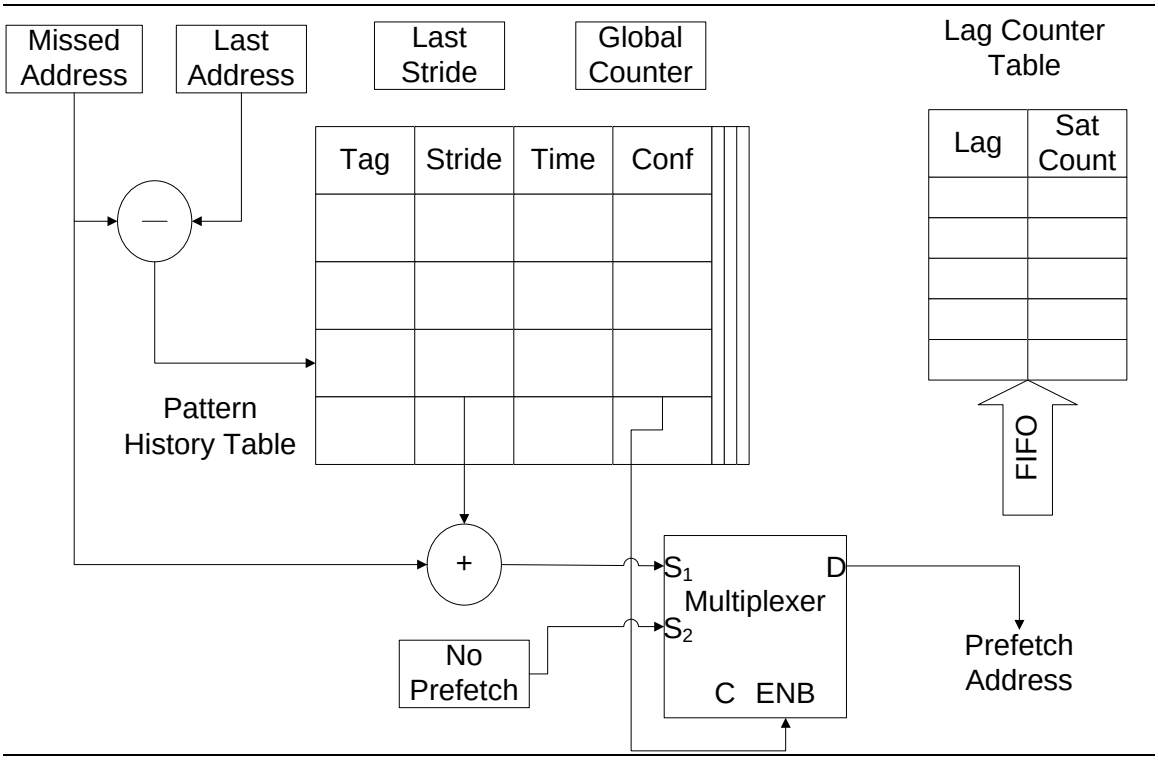


Figure 5-1: Structure of the Differential-only Spectral Prefetcher

To help with the rest of the section, we begin by defining the components of DOSP. Figure 5-1 shows the basic structure and the components of DOSP. Some components have similar nomenclature as those of the spectral prefetcher but there exist key differences between the algorithms governing both these prefetchers. At this point we would like to formalize the definitions using Figure 5-1 as an aid.

The three main components of DOSP are:

- *Pattern History Table*: The Pattern History Table (PHT) stores the stride correlation pairs. This is a set associative table which is indexed by the last observed stride between two missed addresses. Each entry of the PHT has 4 fields. The first two fields store the correlated pair. The *time* field maintains the position of the of the

corresponding stride pair *i.e.* it maintains the "time" when the pair was last present in the stream. Finally, the *conf* field tells whether the pair is part of the pattern or not.

- *Global Counter*: Global Counter (GC) is a continuous counter of miss events. Upon overflow GC is reset, making it a modulus count. During allocation of an entry in the PHT, the GC value is recorded in the time field. Whenever the reappearance of the stride pair is detected, GC is used to calculate the recurring distance using the previously stored time value.

- *Lag Counter Table*: Lag Counter Table (LCT) is used for maintaining information on how many stride pairs share the same recurring distance. The LCT is organized as a FIFO with each entry having two fields: *lag* and *saturation counter*. As reappearance of the stride pair is detected, an associative search of the LCT is done to see if an entry with the same lag (recurring distance) is already present. If so, the count for the entry is incremented. Otherwise, a new entry is allocated.

The other components like the last address and last stride contain the most recently observed address and stride, respectively. These registers are used for indexing and updating the PHT.

5.2.2 Operations of the Prefetcher

The operation of DOSP consists of two basic functions: *lookup* and *update*. Lookup is a straightforward operation to decide whether to perform a prefetch when a miss event occurs. While the update operation is the process of either creating a new entry within the PHT or checking if an existing entry is now confident. In order to understand these operations, we need to discuss four issues: i) how the prefetch address is

generated and issued, ii) how reappearance of the stride pair is detected, iii) how the recurring distance is calculated and iv) how the stride pair are validated as part of the pattern. The first issue covers the lookup operation while the second, third and fourth issues cover the update operation. These issues are described as follows:

- *How the prefetch address is generated and issued?* For generating the prefetch address, first the current stride is calculated from the current missed address and the last observed address. This stride is then used as a PHT index to locate the PHT set. Second, from the PHT set, the entry tagged with the current stride is selected and its stride field value is added to the current missed address to generate the prefetch address. Finally, the decision for issuing the prefetch address depends upon the *conf* bit of the PHT entry. If the conf bit is set, the prefetch is issued otherwise the prefetch is dropped.

- *How reappearance of the stride pair is detected?* Reappearance is detected in the update operation, where the PHT is restored with the current missed stride for maintaining an up-to-date history. In this operation, first the last observed stride is used as a PHT index to locate the associated PHT set. Second, from the PHT set, the entry tagged with the last observed stride is selected and its stride field is matched with the current stride. If both the stride field value and the current stride are the same, we say reappearance has taken place. These operations are shown in the shaded blocks of Figure 5-2, where a flow diagram of the update operation is depicted.

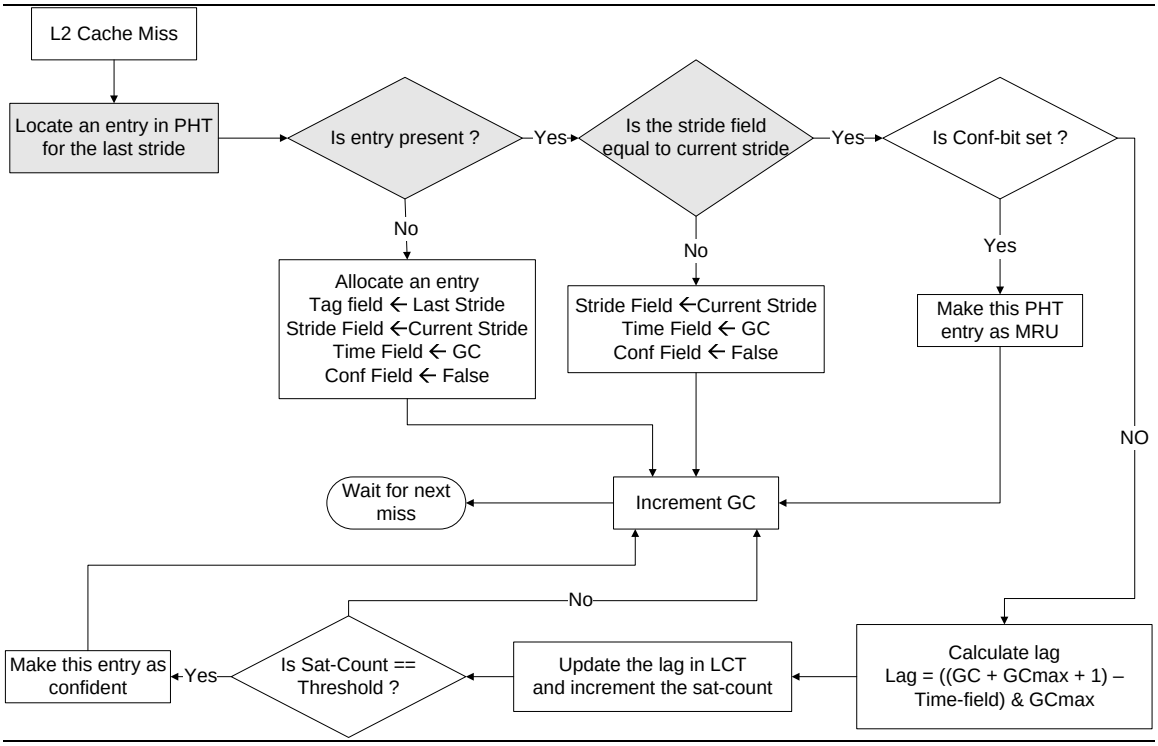


Figure 5-2: Flow diagram of DOSP update operation

- *How the recurring distance is calculated?* Whenever the reappearance of a stride pair is detected, GC is used to calculate the recurring distance (lag) for this pair, using the previously recorded time value. Due to the modulus nature of these values, recurring distance is calculated as: $recurring\ \ distance = ((GC + GC_{\max} + 1) - time)\ \&\ GC_{\max}$, where $GC_{\max}$ is the maximum value of GC. This modulus technique prevents PHT entries from becoming stale upon GC overflow, allowing for continual analysis.

- *How stride pair validated as part of the pattern?* As reappearance of the stride pair is detected, an associative search of the LCT is done to locate an entry with the same recurring distance. Then the *saturation count* field of the LCT entry is incremented. If the value of the saturation count becomes equal to a certain threshold, the stride pair is detected as part of the pattern.

As can be seen from Figure 5-1 and Figure 5-2, there exists key differences between the algorithms of SP and DOSP. First, unlike SP, DOSP has a single table for detection and prediction of pattern. This eliminates the need for expensive update operation of SP, where information is transferred from detection unit to the prediction unit. Second, DOSP is not wasteful and uses continuous analysis for detecting pattern. Thus, the entries of the correlated pair are never cleared rather they are overwritten by newer entries. This allows for the early detection of the high frequency pattern while the remaining portion of the pattern still exist in the PHT for further analysis.

A(0), B(1), ***R1***(2), C(3), D(4), E(5), F(6), ***R2***(7), A(8), B(9), C(10), D(11), E(12), F(13), ***R3***(14), A(15), B(16), C(17), D(18), E(19) ...

Figure 5-3: Example miss stream

We illustrate operations of DOSP with the help of an example miss stream shown in Figure 5-3. This is the same miss stream with which the operations of SP were explained. Here addresses *A, B, C, D, E* and *F* are part of the pattern while *R1, R2* and *R3* are the unordered addresses. For simplicity, we once again assume that the threshold value for the saturation counter is 1. Figure 5-4(a) shows the condition of DOSP after observing first five missed addresses. Similar to SP, DOSP tracks the arrival records of the correlated stride pairs by passing the value of GC in the *time* field of the PHT entry. As shown, the last observed address and stride are *E* and *E – D*, respectively. Here the last address is used for generating the stride while last stride provides the index for updating the PHT. The state of DOSP after missed address *E* arrives for the second time is shown in Figure 5-4(b). When missed address *E* arrives for the second time, the values of last observed address and stride are *D* and *D – C*, respectively. As there was an entry present in the PHT for the correlated pair of *D – C* & *E – D*, the control logic detects

reappearance and allocates an entry in the LCT to track the recurring distance observed in the miss stream. As the saturation counter in the LCT entry becomes equal to the threshold value of 1, the controlling logic assumes that pair $D - C$ & $E - D$ is part of the pattern and sets the conf bit of the particular PHT entry. Finally, the condition of DOSP after missed address D arrives for the third time in the miss stream is shown in Figure 5-4(c). In the meantime the correlated pairs: $E - D$ & $F - E$, $B - A$ & $C - B$, and $C - B$ & $D - C$ reappear by the recurring distance of 7 and are all detected as part of the pattern.

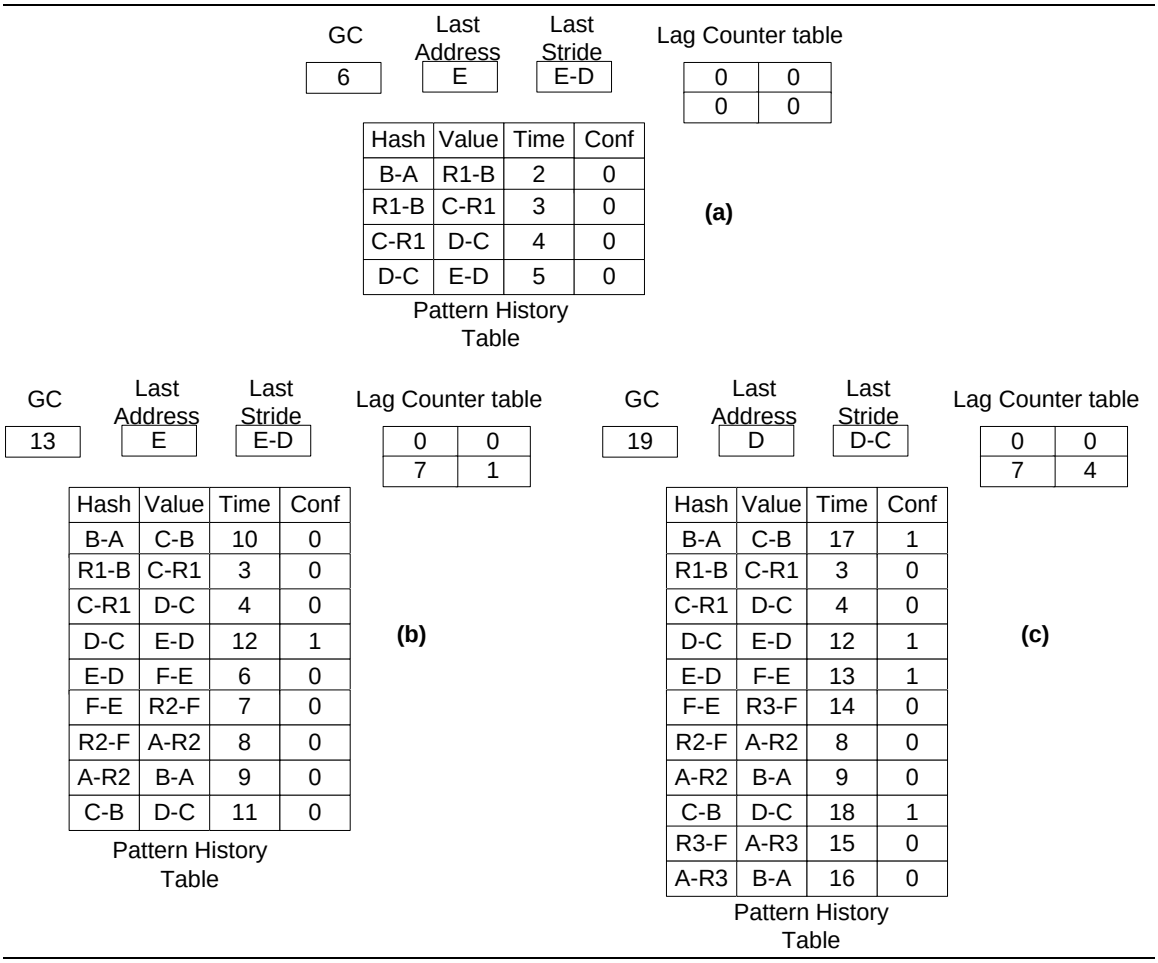


Figure 5-4: Pattern detection example of DOSP

Similarly, the pattern prediction example of DOSP is shown in Figure 5-5. As shown, missed address E arrives for the third time in the miss stream and the current

stride is calculated with the help of the last observed address *D*. Since there is a confident entry for this stride (*E* – *D*) in PHT, a prefetch address is generated by adding the stride field value (*F* – *E*) with the current missed address *E*.

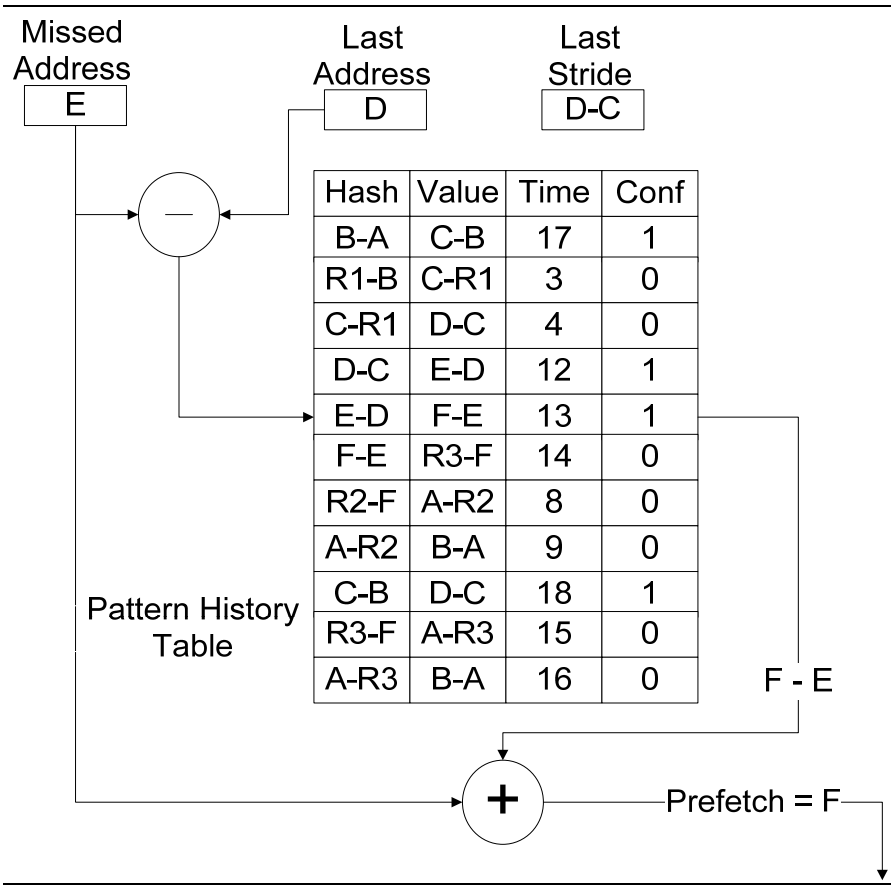


Figure 5-5: Pattern prediction example of DOSP

5.2.3 Correlation Queue for Generating Timely Prefetches

In this subsection, we present a simple technique for generating timely prefetches. The purpose of the correlation queue is to enable correlation of non-adjacent strides. Figure 5-6 shows the structure of the correlation queue. Since DOSP operates in the differential domain, two FIFO queues are maintained: an address queue and a stride queue. The address queue is required for stride generation while the stride queue maintains the history of these strides. Whenever a miss arrives, the current stride is

generated using the tail entry of the address queue, as shown in Figure 5-6. The current stride is then passed to the PHT with the tail of the stride queue to create a pair with a *correlation depth* of *n*, where *n* is the size of the FIFO queues. This distance enables timely prefetching because a lookup match will trigger a prefetch of an address *n* miss events into the future.

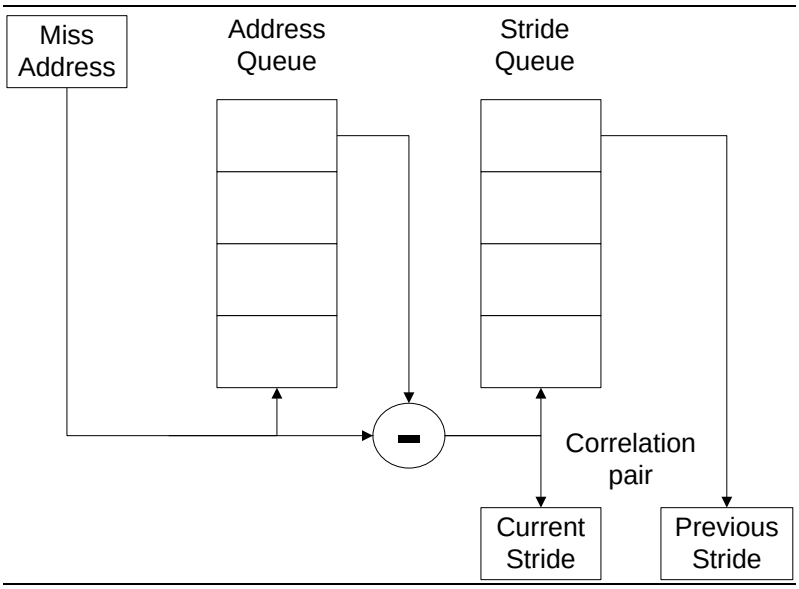


Figure 5-6: Structure of the Correlation Queue

Correlation queue has two advantages over TCzones. First, it correlates non-adjacent addresses in the global miss stream and can be effective for applications that exhibit non-uniformity (where few cache sets are accessed heavily while other sets remain underutilized). Because TCzones localize the global miss stream for fixed number of sets (or set), they are vulnerable in scenarios when all misses are targeted for one of the zone. Second, partitioning memory into TCzones require a dedicated resource (like PHT) and therefore brings up the concerns for complexity in design and extra ports that may be required to update or predict for different zones. Since correlation queue links

two non-adjacent addresses in the global miss stream, there is no need for dedicated resources.

Similar to the behavior of TCzones, larger correlation depths of the queue can lead to highly speculative prefetches. We need to place a threshold on this depth to limit these speculative prefetches. But the question then arises how deep is deep enough, and can memory bandwidth being allocated to these speculative prefetches be better utilized elsewhere. In Section 5.3, we will select the best correlation distance configuration for DOSP with the aid of timing simulations.

5.3 Sensitivity Analysis of the Differential-only Spectral Prefetcher

In this section, we want to develop a sense of how the performance of DOSP changes when two key aspects the depth of the correlation queue and the size of PHT, are varied. For this purpose, we vary the parameter under study and idealize the other. It is important to isolate the contribution made by each component to properly determine an efficient configuration. The base machine used for these experiments is the same as the one described in Chapter 2. For these experiments, we execute only memory-intensive benchmarks on our dual-core base processor. Thus, the core, which is executing the application, solely uses the L2 cache and the lower levels of memory hierarchy. The configuration of other parameters of DOSP selected for these experiments is: i) an 8-entry LCT whose saturation counter saturates at a threshold of 3 and ii) a 6-bit GC.

5.3.1 Depth of the Correlation Queue

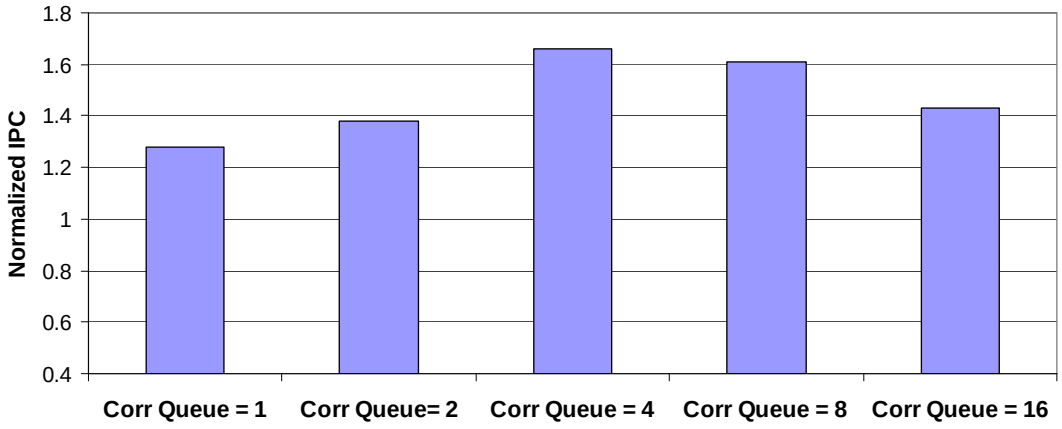


Figure 5-7: Impact of varying the correlation depth on DOSP performance

For this experiment, we varied the depth of the correlation queue to find the optimal size for the general case. Figure 5-7 shows the impact of varying the depth of the correlation queue. Here five different correlation depth sizes are studied: 1, 2, 4, 8 and 16 for 16384 entry 4-way PHT. The IPC results presented are the average gain across the memory-intensive benchmarks for a given configuration of the correlation queue. The detailed results of this experiment are provided in Table 5-1and Table 5-2. The first table shows the results for coverage, accuracy and IPC gains for all the benchmarks while the second table presents the timelines results. Next we present the observations that we can make from these results:

- First, for all the benchmarks, we see that performance increases as we increase the depth of the correlation queue. The reason is that the timeliness of the issued prefetch request improves with the depth of the correlation queue. This can be confirmed from Table 5-2.

- Second, continually increasing the depth actually degrades the performance. The reason is that predictability of the addresses decreases as the correlating distance among them is increased. At a correlation distance of one, adjacent elements are coupled which are likely to recur together. As the depth of the correlation queue increases, however, the correlation pairs become more speculative and thus harder to predict. This can be confirmed in Table 5-1 where it is shown that coverage and accuracy decreases as the queue depth increases.

Table 5-1: Impact of varying the Correlation Queue depth on DOSP performance

Benchmarks	Correlation Queue depth = 1		
	coverage	accuracy	IPC Gain
ammp	99.46%	99.40%	2
mcf	44.99%	98.37%	1
mgrid	87.43%	99.95%	1.64
mst	3.96%	94.06%	1.04
swim	71.13%	96.39%	1.23

Benchmarks	Correlation Queue depth = 2		
	coverage	accuracy	IPC Gain
ammp	99.29%	99.75%	2.5
mcf	41.33%	96.49%	1
mgrid	87.40%	99.91%	1.93
mst	3.80%	93.99%	1.04
swim	70.20%	96.94%	1.38

Benchmarks	Correlation Queue depth = 4		
	coverage	accuracy	IPC Gain
ammp	99.06%	99.59%	4
mcf	41.01%	93.86%	1.33
mgrid	83.68%	99.83%	2.35
mst	3.78%	93.33%	1.04
swim	69.53%	96.94%	1.63

Benchmarks	Correlation Queue depth = 8		
	coverage	accuracy	IPC Gain
ammp	86.93%	99.93%	4
mcf	35.45%	93.20%	1.25
mgrid	82.99%	99.72%	2.34
mst	3.53%	89.02%	1.04
swim	58.41%	96.17%	1.53

Benchmarks	Correlation Queue depth = 16		
	coverage	accuracy	IPC Gain
ammp	80.32%	99.36%	2.5
mcf	28.48%	83.93%	1.16
mgrid	77.60%	99.47%	2.15
mst	3.28%	87.98%	1.04
swim	50.35%	92.31%	1.38

Table 5-2: Impact on the timeliness of Differential-only Spectral Prefetcher

Benchmarks	Correlation Queue depth = 1		
	timely	acceptable	poor
ammp	51%	0%	49%
mcf	35.84%	7.55%	56.61%
mgrid	52.40%	6.24%	41.37%
mst	97.51%	0.28%	2.21%
swim	26.39%	6.97%	66.64%

Benchmarks	Correlation Queue depth = 2		
	timely	acceptable	poor
ammp	66.84%	0%	33.16%
mcf	36.07%	13.46%	50.46%
mgrid	72.51%	0.25%	27.24%
mst	98.57%	0.53%	0.9%
swim	45.01%	6.93%	48.05%

Benchmarks	Correlation Queue depth = 4		
	timely	acceptable	poor
ammp	73.11%	0.01%	26.88%
mcf	68.41%	5.80%	25.78%
mgrid	96.79%	0.54%	2.67%
mst	99.5%	0.17%	0.33%
swim	51.55%	7.33%	41.12%

Benchmarks	Correlation Queue depth = 8		
	timely	acceptable	poor
ammp	98.62%	1.15%	0.33%
mcf	95.24%	0.95%	3.81%
mgrid	99.4%	0.34%	0.26%
mst	99.83%	0.03%	0.13%
swim	94.77%	2.18%	3.05%

Benchmarks	Correlation Queue depth = 16		
	timely	acceptable	poor
ammp	100%	0%	0%
mcf	99.82%	0.03%	0.15%
mgrid	100%	0%	0%
mst	99.92%	0.03%	0.05%
swim	99.40%	0.35%	0.25%

The results suggest that a correlation queue with a depth of 4 produces timely prefetches without degrading the coverage and accuracy of DOSP. For the rest of chapter, we assume a 4 entry correlation queue unless otherwise specified.

5.3.2 Size of the Pattern History Table

Figure 5-8 summarizes the average impact of the PHT size on the performance of DOSP. The detailed results of IPC gain for different configurations of PHT are shown in Table 5-3. PHT, as proposed, is set-associative and is accessed using a stride between the

two missed addresses. As a result, if the PHT is not large enough, the prefetcher will have difficulty in detecting a pattern. For this experiment, we selected the correlation queue depth as 4. We varied the associativity of the PHT for four different set configurations: 2048, 4096, 8192 and 16384 sets respectively.

Table 5-3: Performance of DOSP with different configurations of PHT

Bench-marks	2048 entry PHT				4096 entry PHT				8192 entry PHT				16384 entry PHT			
	1-way	2-way	4-way	8-way	1-way	2-way	4-way	8-way	1-way	2-way	4-way	8-way	1-way	2-way	4-way	8-way
ammp	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
mcf	1	1.08	1.08	1.08	1	1.08	1.08	1.08	1.08	1.08	1.08	1.16	1.08	1.08	1.16	1.33
mgrid	2.41	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42	2.42
mst	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04	1.04
swim	1.48	1.63	1.63	1.63	1.55	1.63	1.63	1.63	1.63	1.63	1.63	1.63	1.63	1.63	1.63	1.63

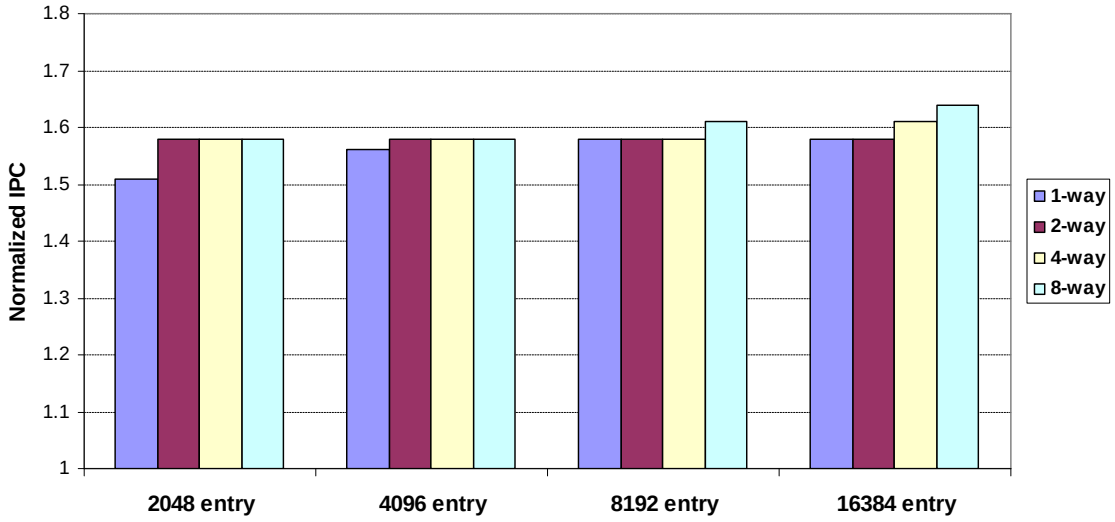


Figure 5-8: Impact of varying the PHT sizes

The results show that 2048 set, 2-way associative PHT is sufficient for detecting pattern for most of the memory intensive benchmarks. The only exception is *mcf* whose performance increases with the increasing size of PHT. These results are expected and

are in direct accordance with the insights gained from Chapter 3, where we showed with the help of autocorrelation that space requirements for most of the benchmarks is small in differential domain. However, the size of the pattern for *mcf* is large (~73462 entry table approximately) and it becomes difficult for DOSP to acquire these patterns with smaller PHT sizes.

From our perspective, the decision as to which PHT configuration is to use is 2048 set, 2-way associative, as it provides the best performance/size tradeoff.

5.4 Evaluation of Differential-only Spectral Prefetcher in Uni-Processor Mode

Table 5-4: Table configuration of GHB-based prefetcher and DOSP

Prefetch Method	Table Configurations	Degree of Prefetch	Size
GHB Based Differential Markov	512 Index Table entries, 4096 GHB entries	4	20 KB
Differential-only Spectral Prefetcher	2048 entry, 2-way Pattern History Table, 8 entry LCT with a two bit saturation counter, 6-bit GC, and correlation queue of depth 4	1	16 KB

In this section, we compare DOSP against Differential Markov GHB implementations. To gauge GHB-based Differential Markov's best performance, we present results for both width and depth prefetching. We maintain a prefetch degree of four for both the GHB configurations. Here degree of prefetch corresponds to fetching multiple cache lines in response to a cache miss [18, 28, 29]. Table 5-4 summarizes the prefetch table configurations for each method. When calculating the table sizes we assumed 26-bit cache line addresses (assuming 32-bit machine and 64 byte cache line size) and 18-bit

stride fields. The sizes in the table are rounded up to the nearest kilobyte. For timing simulation results, we further assume that each access to the prefetch table takes 4 cycles.

GHB-based methods are selected for comparison because they also function in the differential domain and employ a smaller table configuration for acquiring the miss patterns. Furthermore, GHB-based methods are considered to be the best among the recently proposed prefetching mechanisms [32]. This provides us the opportunity to compare DOSP against a most potent competitor.

Before we can analyze the results for GHB-based prefetching mechanisms and DOSP, we need to understand what causes the performance to vary between these mechanisms in the first place. That is, why the prefetch performance may change when the underlying prefetcher is changed? To understand this aspect, we discuss the characteristics of the prefetchers below.

- DOSP uses spectral prediction to filter the unordered addresses and acquires only the repeating patterns for predictions. In contrast, GHB-based prefetching methods use strict value locality by recording every instance of misses as a potential candidate for predictions. Both these mechanisms can prefetch effectively for applications whose miss pattern remains stable across the miss stream. However, for an irregular access pattern that has both repeating and unordered components, DOSP has an advantage in accuracy over GHB-based prefetching methods. This happens because DOSP not only avoids predictions of unordered addresses but also avoids predictions based on them.

- GHB-based prefetching methods use higher prefetch degree either to increase coverage (in case of width prefetching) or to increase timeliness (in case of depth prefetching). Higher prefetch degree can enhance performance of applications that requires aggressive prefetching but becomes unreasonable when there is limited available bandwidth for issuing prefetch requests. This happens because increasing the prefetch degree beyond a certain point increases the memory traffic exponentially and can delay the demand requests. On the other hand, DOSP uses correlations of non-adjacent addresses for generating timely prefetches and fetches only a single cache line in response to a cache miss. This enables DOSP to generate prefetches effectively even in the case of limited bandwidth.

The overall impact on the performance of the prefetcher will depend on how these individual characteristics interact for a given applications. We discuss this further for individual benchmarks in the following section.

5.4.1 Performance Comparison

Table 5-5 shows the individual performance results for both the memory intensive and the compute intensive benchmarks while Table 5-6 shows the timeliness results for the memory intensive benchmarks. The average IPC improvement results for the memory intensive benchmarks are provided in Figure 5-9. DOSP outperforms GHB based prefetching methods on all benchmarks except *mgrid*. On average, DOSP achieves a 1.57 performance improvement for the memory intensive benchmarks, while GHB-width and GHB-depth show performance improvement of 1.22 and 1.33, respectively. We make the following observations from these results.

- *DOSP is more accurate than GHB-based methods*: The results suggest that DOSP has extremely high prefetch accuracy – more than 93% for all the benchmarks. The impact of accuracy is evident in the results of *mcf* and *mst,* where both the GHB methods degrade the performance by 17% and 3%, respectively. The presence of unordered addresses in the reference stream induces the GHB-based methods to issue additional memory traffic that in turn inflicts performance degradation. In contrast, DOSP filters the unordered addresses and acquire only the repeating patterns for predictions, and hence, shows performance improvements of 8% and 4% for *mcf* and *mst,* respectively. Another application that exhibits a similar kind of behavior is *swim* for which DOSP outperforms both the GHB-based implementations.

- *DOSP is timelier than GHB-based methods*: The performance benefits of DOSP are not primarily due to high prefetch accuracy, but are also dependent on another factor, *i.e.* timeliness of the generated prefetch requests. The results demonstrate that correlating non-adjacent addresses through correlation queue produces timely prefetches than compared to depth prefetching with a prefetch degree of four. This happens because depth prefetching gives priority to adjacent addresses initially and then initiates prefetches for the non-adjacent addresses in an ascending order. Since the prefetch requests are serialized on the bus, the effects of the depth prefetching in bringing the non-adjacent addresses are reduced. This can be observed in the results of *ammp* benchmark where depth prefetching shows a performance gain of 250% while DOSP shows a performance gain of 400%.

The prefetch degree of GHB-based methods does more harm than good. In realistic memory constraints, the additional memory traffic generated by the prefetcher

using a high prefetch degree is often detrimental to performance. Prior research on prefetching assumed much less constrained limits on available memory bandwidth (often unlimited), and have shown that blind selection of a constant prefetch degree improves performance.

Table 5-5: Detailed Performance results for GHB-based methods and DOSP

Results for Memory-intensive Benchmarks									
Benchmarks	**GHB-width**			**GHB-depth**			**DOSP**		
	coverage	accuracy	IPC Gain	coverage	accuracy	IPC Gain	coverage	accuracy	IPC Gain
ammp	99.75%	99.72%	2	99.48%	99.58%	2.5	99.06%	99.59%	4
mcf	21.90%	26.77%	0.83	27.92%	29.44%	0.83	18.45%	91.89%	1.08
mgrid	91.45%	99.15%	1.61	94.32%	89.40%	2.45	83.68%	99.83%	2.35
mst	13.85%	18.09%	0.97	15.71%	19.36%	0.97	3.78%	93.33%	1.04
swim	50.66%	70.36%	1.38	78.44%	73.97%	1.5	69.53%	96.94%	1.63
Results for Compute-intensive Benchmarks									
Benchmarks	**GHB-width**			**GHB-depth**			**DOSP**		
	coverage	accuracy	IPC Gain	coverage	accuracy	IPC Gain	coverage	accuracy	IPC Gain
art	43.85%	96.40%	1	37.28%	94.83%	1	41.79%	99.57%	1
gcc	62.60%	59.45%	1.02	52.59%	50.22%	1.01	50.99%	98.33%	1.01
parser	77.98%	79.80%	1.10	64.92%	70.65%	1.11	61.77%	97.93%	1.13
twolf	14.24%	64.86%	1	17.40%	65.75%	1	0.02%	100%	1
vpr	13.25%	50.28%	1	11.68%	42.97%	1	0.01%	100%	1

Table 5-6: Timeliness results for GHB-based methods and DOSP

Benchmarks	**GHB-width**			**GHB-depth**			**DOSP**		
	timely	acceptable	poor	timely	acceptable	poor	timely	acceptable	poor
ammp	49.95%	0.27%	49.78%	61.84%	0%	38.17%	73.11%	0.01%	26.88%
mcf	50.17%	4.81%	55.02%	64.36%	3.40%	32.24%	68.41%	5.80%	25.78%
mgrid	53.28%	6.21%	40.51%	82.51%	3.72%	13.77%	96.79%	0.54%	2.67%
mst	81.18%	3.63%	15.18%	85.56%	6.95%	7.49%	99.39%	0.22%	0.40%
swim	37.80%	5.82%	56.37%	43.49%	8.37%	48.14%	51.55%	7.33%	41.12%

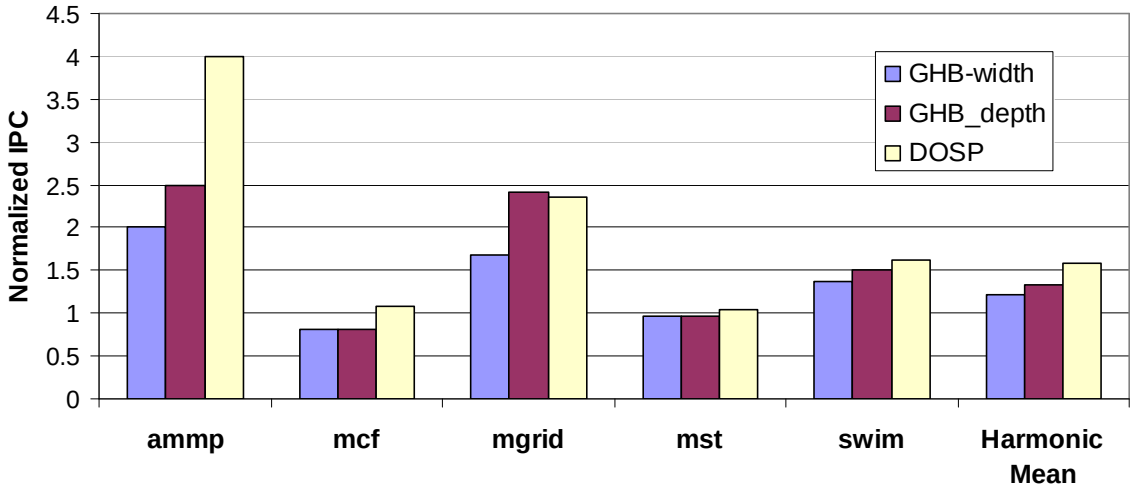


Figure 5-9: IPC gain of GHB-based methods and DOSP

5.5 Evaluation of Differential-only Spectral Prefetcher in CMP Mode

In this section, we explore the performance of DOSP in a dual-core CMP based multiprocessor. It is reasonable to expect that the prefetcher may perform differently when different set of benchmarks are co-scheduled to run on separate CMP cores. For example, a prefetcher may degrade the performance of co-scheduled compute intensive benchmarks by disturbing cache locality while an intelligent prefetching mechanism may help a pair of memory-intensive benchmarks. The task of this section is to develop a sense of how might the prefetcher performance may change when benchmarks of different characteristics are co-scheduled in a CMP environment. For this purpose, we study three different scenarios: (i) when two memory intensive benchmarks are co-scheduled, ii) when a memory intensive is co-scheduled with a compute intensive, and iii) when a pair of compute intensive benchmarks are co-scheduled together.

The experiments presented in this section also provide answers to the question such as: Which is a better prefetching model for a CMP environment – one that intelligently selects what to prefetch or an aggressive mechanism that attempts to bring everything? What are the scenarios where prefetching fails in CMP environment? *etc.*

5.5.1 Experimental Setup

The base machine, over which the speedup numbers are measured, is the same as the one described in Chapter 2. Similar to the uni-processor experiments, we compare the results of DOSP against GHB-width and GHB-depth. To enable prefetching for individual threads, the prefetchers were made to duplicate resources for each thread. In case of DOSP, we only duplicate the correlation queue while PHT is typically shared between the threads. On the other hand, GHB-based mechanisms were completely duplicated among the two threads *i.e.* a separate copy is dedicated for each thread. This gives a slight advantage to the GHB-based prefetching methods. Prefetch table configurations are similar to what we selected for the uni-processor environment and are shown in Table 5-4.

For these experiments, we collected results for 45 different benchmark pairs that were co-scheduled to run on separate CMP cores. To observe the impact of L2 cache prefetching, each benchmark pair is skipped to the first SimPoint (except *mst*) and simulation is terminated once the benchmark pair is executed for 100 million core cycles. We used cycles instead of instructions because different benchmarks have different execution characteristics, i.e. they have different levels of parallelism and some may finish earlier than the others. A threshold of cycles is a better choice for making sure that speedup results are from actual prefetching of the simultaneously executed benchmarks

in a CMP environment. We evaluate the impact on prefetching in terms of changes in three metrics: (i) the *misses per 1000 instructions*, (ii) the *timeliness* of the issued prefetch requests, and (iii) the *throughput* – *i.e.*, the combined progress rate (IPC) of the co-scheduled threads.

5.5.2 Memory Intensive vs. Memory Intensive

In Figure 5-10, we show the performance results when the memory intensive benchmarks are co-scheduled together. The misses per 1000 instructions (miss rates) and the timeliness results are shown in Table 5-7 and Table 5-8, respectively. The miss rates are presented for the base case (without prefetcher) and for three different prefetching schemes. Here the first column under every scheme shows the miss rate of the first thread, while the second column shows the results for the second thread. The speedup results are presented in terms of improvements over the throughput of the base case.

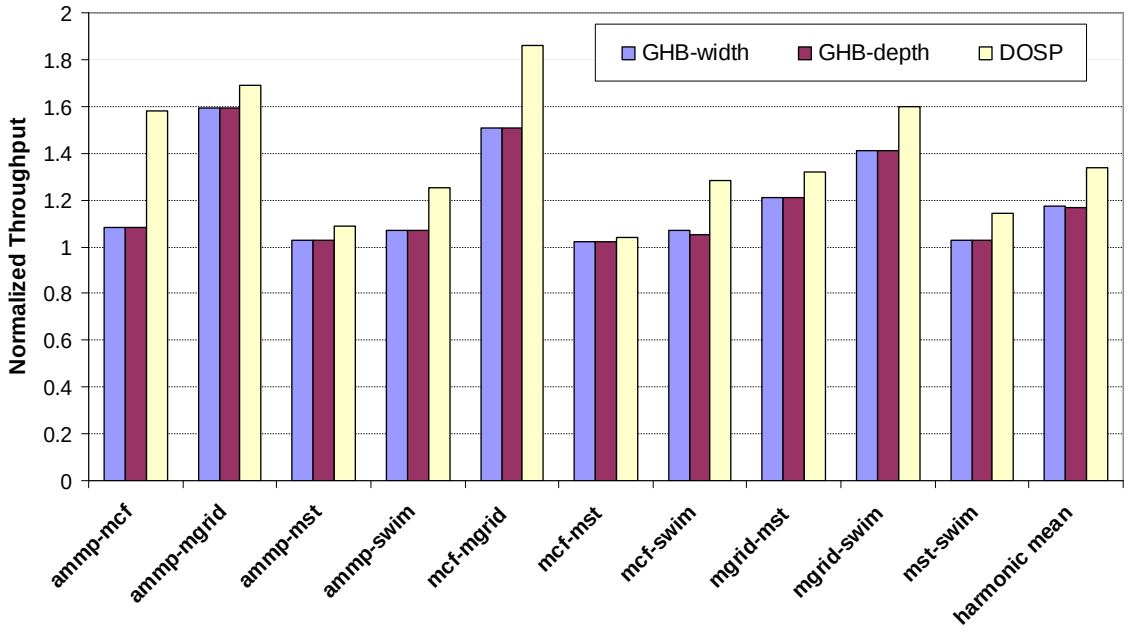


Figure 5-10: Throughput results for memory intensive benchmarks

Table 5-7: Miss rate of co-scheduled memory intensive benchmarks

Benchmark-pair	Base		GHB-width		GHB-depth		DOSP	
	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2
ammp-mcf	89.53	48.55	4.17	41.14	4.25	42.65	5.67	35.69
ammp-mgrid	89.37	3.41	3.51	0.48	3.51	0.48	6.71	1.12
ammp-mst	89.35	6.87	2.93	6.50	2.88	6.51	2.88	6.51
ammp-swim	89.19	12.20	5.71	7.91	5.68	7.91	10.76	5.39
mcf-mgrid	77.16	3.37	51.36	0.55	51.60	0.56	29.11	0.80
mcf-mst	74.67	6.80	50.44	6.55	50.44	6.55	33.74	6.93
mcf-swim	76.40	12.26	54.66	8.05	55.17	8.07	34.50	5.09
mgrid-mst	3.96	6.98	1.24	6.94	1.24	6.95	1.72	6.99
mgrid-swim	3.85	12.27	0.61	7.75	0.61	7.75	1.17	4.49
mst-swim	6.01	12.32	6.23	8.29	6.24	8.27	6.19	4.79

Table 5-8: Timeliness results for co-scheduled memory intensive benchmarks

Benchmark-pair	GHB-width			GHB-depth			DOSP		
	timely	acceptable	poor	timely	acceptable	poor	timely	acceptable	poor
ammp-mcf	62.83%	3.82%	33.34%	63.53%	4.00%	32.47%	75.82%	6.84%	17.34%
ammp-mgrid	47.13%	4.75%	48.12%	47.13%	4.75%	48.12%	80.33%	5.54%	14.13%
ammp-mst	60.07%	4.01%	35.92%	60.06%	4.01%	35.92%	82.74%	5.31%	11.95%
ammp-swim	34.29%	8.00%	57.71%	34.23%	7.68%	58.09%	69.93%	6.58%	23.49%
mcf-mgrid	69.59%	4.84%	25.57%	70.03%	4.75%	25.22%	83.94%	4.91%	11.15%
mcf-mst	87.74%	2.72%	9.54%	87.74%	2.72%	9.54%	88.45%	5.02%	6.53%
mcf-swim	61.72%	7.28%	31.00%	61.74%	7.43%	30.83%	71.26%	6.60%	22.14%
mgrid-mst	66.61%	4.64%	28.75%	66.57%	4.64%	28.79%	97.87%	0.65%	1.48%
mgrid-swim	35.91%	7.2%	56.89%	35.91%	7.2%	56.89%	75.53%	5.14%	19.33%
mst-swim	61.96%	3.20%	34.84%	61.94%	3.16%	34.90%	80.94%	3.17%	15.89%

From Figure 5-10, we can observe that, in general, DOSP outperforms GHB-based mechanisms for all the co-scheduled pairs. On average DOSP achieves a 1.33 performance improvement, while GHB-width and GHB-depth show performance improvement of 1.17 and 1.16, respectively. The performance of DOSP is better than the GHB-based methods primarily due to three important reasons, which are explained as follows.

- DOSP reduces the miss rates for all the memory intensive benchmarks (except *mst*) while GHB-based methods target the ones that exhibit regular behavior in their respective access stream (*ammp* and *mgrid*). This happens because DOSP is more accurate and is able to reduce the load memory latency for benchmarks (*mcf* and *swim*) that exhibit the presence of unordered addresses. In contrast, GHB-based methods reduce the miss rates for these benchmarks but not as significantly as DOSP.

- DOSP is timelier than the GHB-based methods (as shown in Table 5-8). This is also evident from the performance result of highly predictable benchmarks, *ammp-mgrid*, where DOSP shows better performance than GHB-based methods. The reason is that correlating non-adjacent addresses through correlation queue produces timely prefetches than compared to depth prefetching with a prefetch degree of four.

As noted earlier, another shortcoming of GHB-based methods is their prefetch degree, which induces more than one prefetch request for a single miss event. This philosophy of quantity over quality can be detrimental to performance. Overall, the results suggest that DOSP performs better than the GHB-based methods in a CMP environment that executes memory-intensive benchmarks simultaneously.

5.5.3 Compute Intensive vs. Compute Intensive

In Figure 5-11, we show the performance results when the compute intensive benchmarks are co-scheduled together. The misses per 1000 instructions (miss rates) and the timeliness results are shown in Table 5-9 and

Table 5-10, respectively.

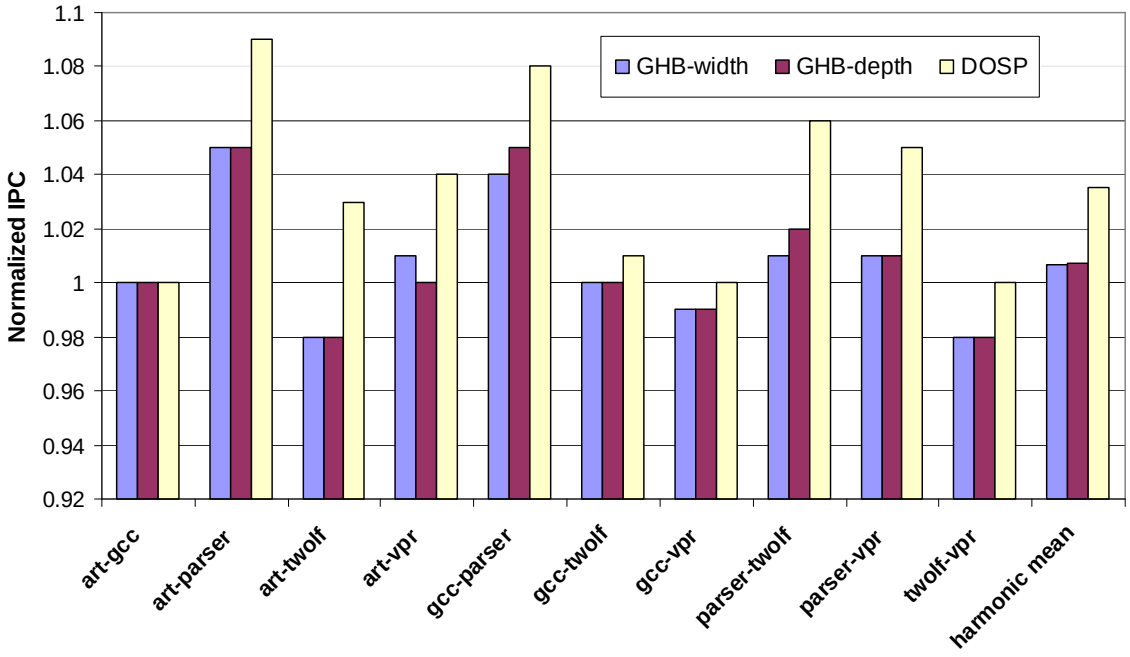


Figure 5-11: Throughput results for compute intensive benchmarks

From Figure 5-11, we can observe that, DOSP outperforms GHB-based mechanisms for all the co-scheduled pairs. On average DOSP achieves a 1.03 performance improvement, while GHB-width and GHB-depth show no performance improvement. We can further observe that DOSP reduces the miss rates for all the pairs considerably when compared to GHB-based methods and is also timelier.

It is worth noting that GHB-based methods inflict performance degradation for pairs: *art-twolf*, *gcc-vpr*, and *twolf-vpr*. This happens primarily because *art*, *gcc*, *twolf*, and *vpr* are marred with unordered addresses in their respective miss stream and GHB-based method, which follow strict value locality, issue ineffectual prefetches for these applications. Moreover, these benchmarks have smaller working set sizes and have a smaller temporal reuse distance; as a result an in-accurate prefetching mechanism can

harm the performance. On the other hand, DOSP never inflicts any performance degradation because it attempts to either avoid predictions (for *twolf* and *vpr*) or intelligently select what to prefetch (for *art* and *gcc*).

Table 5-9: Miss rate of co-scheduled compute intensive benchmarks

Benchmark-pair	Base		GHB-width		GHB-depth		DOSP	
	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2	Misses/1000 instructions for Thread 1	Misses/1000 instructions for Thread 2
art-gcc	3.38	0.81	2.22	0.60	2.40	O.54	1.75	0.61
art-parser	2.87	2.92	3.06	1.70	3.06	1.70	2.55	1.50
art-twolf	4.24	6.67	3.76	6.88	3.76	6.88	2.77	6.56
art-vpr	6.15	4.67	4.22	4.80	4.65	4.74	3.40	4.60
gcc-parser	0.47	1.16	0.34	0.53	0.33	0.53	0.37	0.42
gcc-twolf	0.62	0.65	0.44	0.70	0.44	0.70	0.44	0.66
gcc-vpr	0.53	0.45	0.39	0.47	0.40	0.47	0.37	0.45
parser-twolf	1.47	1.01	0.75	1.41	0.74	1.29	0.65	1.40
parser-vpr	1.42	0.67	0.68	0.90	0.70	0.91	0.63	0.91
twolf-vpr	1.98	1.47	2.15	1.50	2.15	1.50	1.99	1.50

Table 5-10: Timeliness results for co-scheduled memory intensive benchmarks

Benchmark-pair	GHB-width			GHB-depth			DOSP		
	timely	acceptable	poor	timely	acceptable	poor	timely	acceptable	poor
art-gcc	55.47%	1.94%	42.59%	54.49%	1.82%	43.69%	58.46%	5.28%	36.26%
art-parser	40.65%	2.2%	57.15%	40.65%	2.2%	57.15%	75.91%	4.10%	19.99%
art-twolf	70.12%	1.66%	28.22%	70.12%	2.11%	27.77%	88.80%	3.02%	8.18%
art-vpr	70.73%	1.85%	27.42%	69.49%	1.77%	28.74%	88%	3.80%	8.20%
gcc-parser	49.74%	1.22%	49.04%	48.97%	1.12%	49.91%	57.64%	5.32%	37.04%
gcc-twolf	72.82%	1.02%	26.16%	72.52%	0.87%	26.60%	74.39%	9.06%	16.54%
gcc-vpr	71.13%	0.82%	28.05%	71.58%	0.92%	27.50%	72.03%	9.12%	18.85%
parser-twolf	73.77%	0.80%	25.43%	73.76%	1.53%	24.71%	84.66%	7.29%	8.05%
parser-vpr	74.87%	1.27%	23.86%	74.88%	1.25%	23.87%	85.41%	6.11%	8.48%
twolf-vpr	99.59%	0.06%	0.35%	99.59%	0.11%	0.30%	100%	0%	0%

5.5.4 Memory intensive vs. Compute Intensive

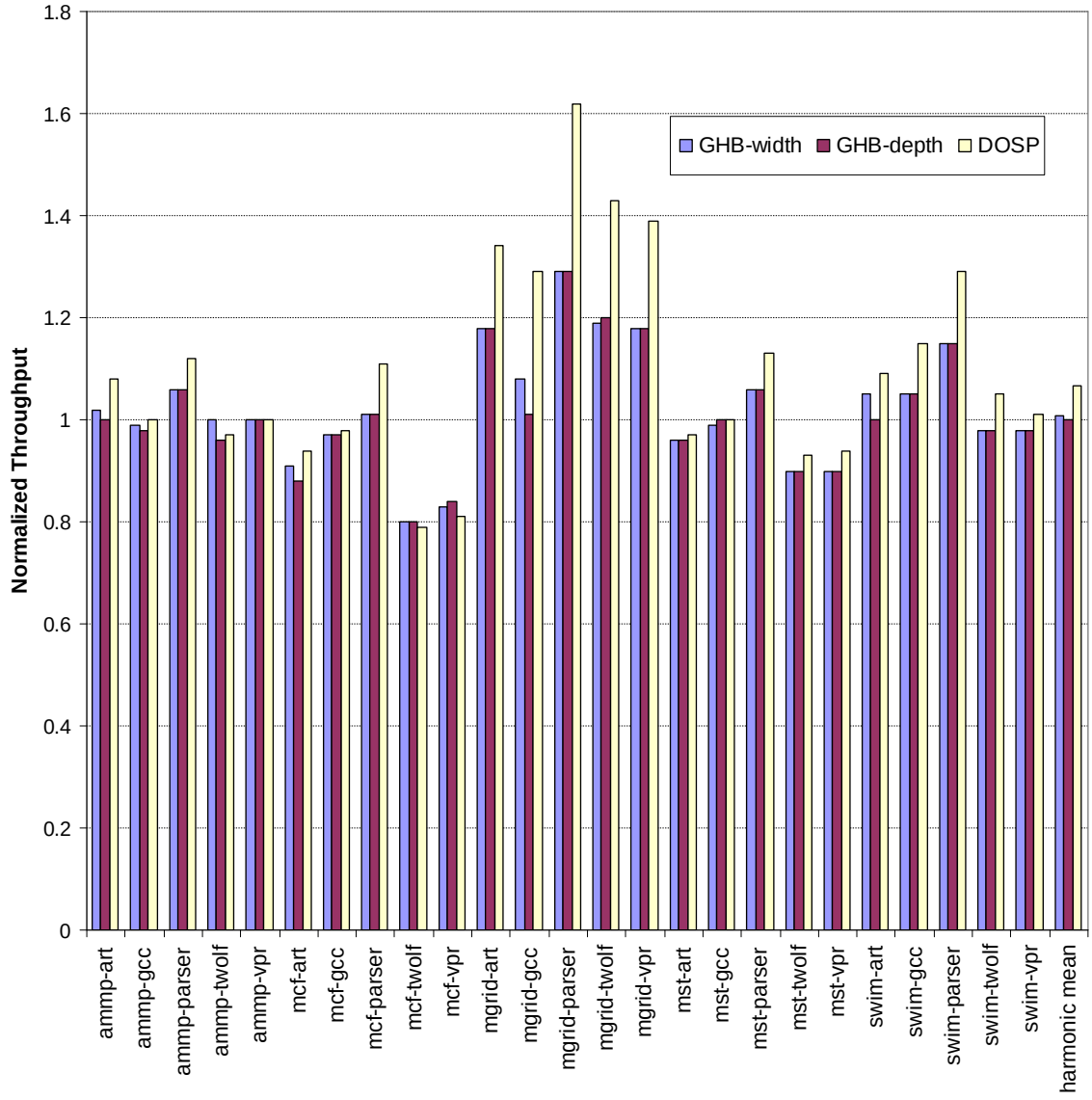


Figure 5-12: Throughput results for mixed (memory and compute) pairs

In Figure 5-12, we show the performance results when the memory intensive benchmarks are co-scheduled with compute intensive benchmarks. As shown, DOSP outperforms GHB-based mechanisms for all the co-scheduled pairs (except *mcf-twolf* and *mcf-vpr*). On average DOSP achieves a 1.06 performance improvement, while GHB-width and GHB-depth show performance improvement of 1.01 and 1.00, respectively.

It is worth noting that DOSP inflicts performance degradation for the following pairs: *ammp-twolf, mcf-art, mcf-gcc, mcf-twolf, mst-art, mst-twolf,* and *mst-vpr*. This primarily happens because of the two reasons. First, the compute intensive benchmarks have a smaller temporal reuse distance compared to memory intensive benchmarks. As a result, demand fetch model through the use of LRU replacement policy ensures that compute intensive applications have a larger share in the caches. In contrast, prefetchers usually favor the application that show higher number of misses in the miss stream, and hence, changes the cache space configuration. This in turn results in performance losses for the compute intensive benchmark. In general, compute intensive application have a higher IPC rates (because they have lower cache misses) and any performance degradation for them can have a huge impact on throughput. Thus, prefetcher helping only a memory intensive benchmark can lower the throughput of the co-scheduled pair. Second, most of the compute intensive benchmarks present in the above shown pairs exhibit mainly unordered addresses in their respective miss stream. As a result, prefetcher faces difficulty in helping them and mostly helps the memory intensive benchmark.

The GHB-based methods also suffer from the above discussed issues, but they also face a larger issue of predicting unordered addresses that often results in ineffectual prefetches.

5.6 Chapter Summary

In this chapter, we proposed a simplified implementation of spectral prediction: *Differential-only Spectral Prefetcher* (DOSP) that detects only the stride patterns within the global miss stream. DOSP evolved through the continuous simplification of the SP

algorithm. In this process, we identified the limitations of SP and formulated their individual solutions. This in turn laid a foundation for DOSP, which not only uses dramatically less size but also provides better performance. There exist several differences between the design of DOSP and SP and some of the major ones are listed as follows:

- Unlike SP, DOSP uses a continuous analysis for acquiring the patterns – *this eliminates the problem of high frequency latching*

- DOSP uses a single table for pattern detection as well as prediction – *this in turn eliminates the need for costly operation like associative searches and pattern transfer.*

- DOSP primarily function in the differential domain – *this exploits the reduced size benefits that differential domain have to offer.*

- Finally, DOSP uses a correlation queue for correlating non-adjacent addresses to generate timely prefetches – *this further removes the complexity associated with employing TCzones.*

When added to an aggressive superscalar processor with large caches and branch predictor, DOSP produce performance improvements in the 4% to 400% range on memory intensive benchmarks. We also explore the performance of DOSP in a dual-core CMP based multiprocessor environment. The experimental results show that DOSP outperforms the previously proposed schemes even in the CMP environment with an average throughput gain of 10% (shown in Figure 5-13).

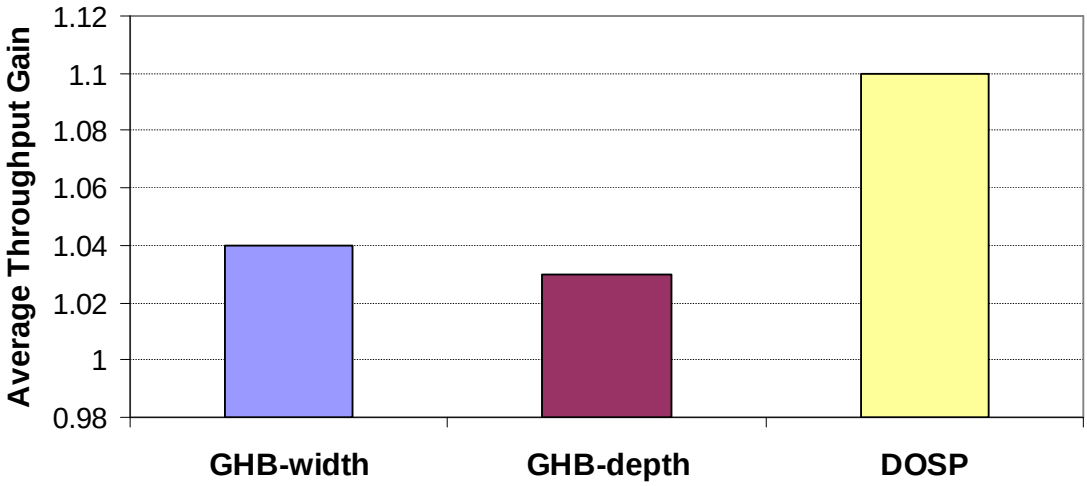


Figure 5-13: Average throughput gain for different prefetching schemes

Chapter 6 Related Work

Our work builds on the contributions of much of the past research from the last three decades. In this chapter, we will review some of the related work in order to place our work in the context of other research. The chapter is organized into four sections: compiler-based data prefetching, specialized hardware prefetching, correlation-based hardware prefetching, and thread based prefetching.

6.1 Compiler-based Data Prefetching

Compiler-based prefetching inserts explicit prefetching directives in the code to fetch data into the cache. Most modern microprocessors have a *prefetch* instruction, which can be used by the compiler to initiate prefetches. These instructions are often used within loops responsible for large array calculations in scientific applications. The references produced by these loops are highly predictable and through static analysis prefetch instruction can be placed inside the loop to fetch data for future loop iterations. Based on this approach, Mowry *et al.* [27] were the first to propose compiler algorithms that automatically added prefetch instructions during an optimization pass of the compiler. Bernstein et al. [3] and Sanathanam et al. [35] used the original algorithm for running

scientific codes on the real machines and showed that performance can be increased using explicit prefetch instructions.

Similar attempts for general applications [8, 23, 25] resulted in insignificant performance gain because these application are far more irregular than their scientific counterparts. Moreover, many general application exhibit high temporal locality and thus decreases the benefit of prefetching. There are three shortcomings of compiler based prefetching. First, the prefetch instruction not only uses the execution resources, but also needs a register as a source for fetching a cache line. The address value for a future request is passed to the register by other instructions. Since there is a limited register space with the compiler, the use of prefetch instruction usually results in additional register spills in the code. Second, aggressive compiler based method may end up increasing the code size of the actual program resulting in instruction cache misses. Finally, the lack of run time information makes difficult for the compilers to judiciously decide what to prefetch.

6.2 Specialized Hardware Prefetching

We call a hardware prefetching model *specialized* if it targets specific access pattern in the memory reference behavior. The simplest among these prefetchers is Sequential prefetching [37] that exploits spatial locality present in the applications. These prefetchers simply prefetch a forward or backward cache line for either a hit access or miss access of the corresponding line. The problem with this approach is that prefetch may not be timely enough for hiding the demand request. To overcome this problem, degree of prefetch – fetch multiple cache lines for a single miss event – was proposed

[12]. This can introduce additional memory traffic and can disturb the demand fetch locality of the caches for the phases of code that does not have any spatial locality.

Similar to the above ideas, Jouppi proposed stream buffers [19] to improve cache performance for sequential missed accesses. The stream buffers are organized as FIFO structure where the prefetched cache lines are placed. If a demand request is served by the stream buffer, the corresponding line is passed to the cache, and all the other lines are moved up in the FIFO queue. Stream buffers never bring a line directly into the cache and thus prevent cache pollution. Palacharla and Kessler extended the effectiveness of stream buffers by allocation filtering and non-unit stride detection mechanisms [30]. Other stride prefetching mechanism includes Chen and Baer stride prefetcher that correlates PC of memory instruction [8].

As with compiler based prefetching, the hardware prefetching discussed above focus on array references. There are some other mechanisms that target the linked-data-structure references. First among them was proposed by Harrison and Mehrotra [14] that extended Chen and Baer's arbitrary stride prefetcher for prefetching data objects connected via pointers. In this scheme, a field was added to the original design of stride prefetcher, which enabled detection of indirect reference strides arising from linked lists or indirect array accesses. Another hardware mechanism designed to prefetch data structure connected by pointers was proposed by Roth et al. [33]. This mechanism has the ability to run ahead the pointer intensive application to mask the prefetch latency. Similarly, Cooksey *et al.* [11] proposed Content-directed Data Prefetching, which prefetches the connected linked data structure elements by examining the data contents of the missed cache line.

These specialized prefetching mechanisms are effective for specific access patterns, but have limited applicability across wide range of application programs.

6.3 Correlation-based Prefetching

A desire for a generic prefetcher prompted the development of the correlation-based prefetching, which is able to remember complex patterns present in the reference stream of general applications. The idea behind correlation prefetching is to use previously seen missed addresses to predict the future missed addresses.

Charney and Reeves [6] were the first to propose a correlation prefetching scheme for L1 miss reference stream. In this scheme, a hardware cache maintains parent–child pair information, where parent corresponds to the first cache line and child corresponds to the cache line accessed right after the first cache line. Since it is possible that a given missed cache line may be followed by one of several different lines depending upon the control flow or data flow, Joseph and Grunwald [18] proposed a Markov model for prefetching. This maintains a set of states that are connected with transition arcs denoting transition from one state to another. In the case of data-prefetching, states denote the cache lines and transition arcs denote the probability of transition from one cache line to another.

To address timeliness, Lai et al. [22] proposed hardware-based deadblock predictor, which keeps history of both PC traces and memory addresses. In this scheme, the prefetcher predicts when the cache line becomes “dead” and prefetches for that given cache set. Another approach that predicts for a given cache set is Hu et al’s Tag

Correlating Prefetcher [15], which exploits correlation among cache tags rather than whole cache line addresses.

Recently, Nesbit and Smith proposed Global History Buffer [28] for maintaining address correlations. They demonstrated that a circular buffer can successfully eliminate stale data information when compared to a table-based approach.

While all the correlation based schemes attempt to predict repeating patterns present in the miss stream, they are often associated with lower prediction accuracy. This happens because they use strict value locality by recording every instance of misses as a potential candidate for prediction, and hence, are unable to differentiate between the repeating contexts and unordered addresses.

6.4 Thread based Prefetching

The emergence of multithreaded processors [41] has led to thread based prefetchers, which execute a reduced version of the original program ahead of the program execution to avoid cache misses. These techniques, typically, extract a program slice for each critical load instructions that are responsible for the vast majority of the memory stall cycles. The program slice is executed in another thread context and is often termed as the *speculative thread.*

Several thread based prefetching mechanisms have been proposed and are usually differentiated based on how they extract slices from the original program. Zilles and Sohi proposed Speculative Slices [43] which is primarily software based technique designed for use in SMT processors. Similarly, Roth and Sohi proposed Data Driven

Multithreading [34], in which speculative threads are constructed statically and execute on idle thread contexts to prefetch for future memory accesses. Contrary to the above discussed schemes, Collins *et al.* proposed Dynamic Speculative Precomputation [10] that extracts the speculative slices dynamically and is implemented on top of an in-order multithreaded processor.

Another dynamic approach is Slipstream Processors [39], in which non-speculative thread of the program runs alongside a shortened, speculative thread. Outcomes of critical instructions in the speculative thread are passed to the original program, providing speedup if the speculative outcome is correct. This work primarily focused on a chip-multiprocessor.

A potential shortcoming of thread based prefetching is that cache misses can prevent the speculative thread from making progress faster than the original program. Thread prefetching can fail for applications that use linked-data-structures traversals in their algorithms.

Chapter 7 Conclusions

In this chapter, we first present a summary of this thesis and then discuss the various directions in which this work can be extended.

7.1 Thesis Summary

This dissertation introduces a novel concept and two novel mechanisms. The concept is *spectral prediction*, a technique for stepping outside the strict value locality framework of traditional correlated prefetchers. The mechanisms are *Spectral Prefetcher* (SP) and *Differential-only Spectral Prefetcher* (DOSP), implementations of spectral prediction, which act as a natural extension to the correlation-based prefetchers.

7.1.1 Spectral Prediction

Performance of a correlation-based prefetcher decreases in the presence of unordered addresses, which recur in a random or uncorrelated manner in the reference stream. We observed that unordered addresses are very pervasive and are present in the access stream of applications either due to control flow irregularities or due to dynamic transformations of linked-data-structures. Predictions of these addresses or predictions based on them results in ineffectual prefetches, which can lead to higher memory

utilization and performance losses. Spectral prediction is a way of using recurrence distance information to prune out unordered addresses from the access stream.

Generally, the elements (addresses) of the pattern follow fixed frequencies and thus collectively reappear with the same recurring distance in the reference stream. Spectral prediction exploits this property – collective reappearance – for pattern detection and in a way tunes to the frequency (or frequencies) of the pattern.

7.1.2 Implementations of Spectral Prediction

An implementation of spectral prediction has three components. The first component maintains the arrival records of the previously seen correlated pairs in the reference stream. The second component maintains the continuous count of the miss events and is used for calculating the recurring distances of the reappearing pairs. Finally, the third component maintains the previously observed recurring distances with which the correlated pairs are reappearing in the stream.

This dissertation proposes two prefetching architectures that implements spectral prediction by using the above described components. Spectral Prefetcher (SP) is our first proposed implementation of spectral prediction. SP, as proposed, divides the memory address space into Tag Concentration Zones (TCzones) for generating timely prefetches and detects either the pattern of tags (higher order bits) or the pattern of strides (differences between the consecutive tags) within each TCzone. The prefetcher dynamically determines whether the pattern of tags or strides will increase the effectiveness of prefetching and switches accordingly. SP was designed as a proof-of-concept and provided productive insights for designing a more elegant implementation:

Differential-only Spectral Prefetcher (DOSP), which is resource-efficient and offers better performance. DOSP detects only stride patterns within the global miss stream and has the unique property of tying two non-adjacent strides for generating timely prefetches.

7.1.3 Performance Potential of Spectral Prefetchers

The results of our performance evaluation, which may be described as preliminary, are encouraging. When added to an aggressive superscalar processor with large caches and branch predictor, SP and DOSP produce performance improvements in the 4% to 400% range on memory intensive benchmarks. Additionally, using a set of co-scheduled pairs of benchmarks on a dual-core CMP, we show that a 16KB on chip implementation of DOSP provides an average throughput improvement of 10% and at best 86%.

The implementations of spectral prediction, SP and DOSP, have a positive outcome and achieve their stated goal of reducing load execution latency. Overall, this thesis provides a promising way of applying data prefetching for future microprocessors.

7.2 Future Work

This work is an initial effort in the area of data prefetching using spectral prediction. There is an immense potential for further research in this and the other fields. In the next sections we present three different ways in which the work can be extended further.

7.2.1 Introduce Reconfigurable Degree of Prefetch in DOSP

We have seen that degree of prefetch does more harm than good in case of GHB-based prefetching methods. This happens because different programs have different

access patterns and optimal prefetch degree vary across programs. Moreover, the access patterns may change within a program itself as program goes through various phases of execution. Thus, like any other design parameter prefetch degree can be optimized for good performance.

Future work will employ a dynamic tuning algorithm to decide an optimal prefetch degree of DOSP. This dynamic reconfiguration will not only increase the performance for applications that requires aggressive prefetching mechanisms but will also provide a mechanism for shutting down prefetching completely in those cases where prefetching hurts performance. It will be interesting to study this dynamic reconfiguration in CMP environment, where different threads could be assigned different prefetch degree for higher throughput gains.

7.2.2 Better Management of the Secondary Level Caches

In this thesis, we presented spectral prediction as a means for fetching the pattern into the caches. We can apply converse of this operation *i.e.* removing the cache lines depending upon the information that an address represent an unordered or ordered element. Since unordered addresses occur infrequently in the reference stream, it will be better to replace them as early as possible. We propose a technique where LRU replacement policy should only be maintained for the repeating patterns while the unordered address should be picked randomly form the cache sets. It will be interesting to see if we can maintain two different levels of LRU policies – one for ordered addresses and the other for unordered addresses. .

7.2.3 Branch Predictor based on Spectral Prediction

There are two extremes – one is to represent history in time domain and another is to translate them in frequency domain using a spectral technique like Fourier analysis. Both have their own advantages and disadvantages. History in time domain is smaller but the implementation of mechanism is easier. On the other hand, history information can reach to very large scale in the frequency domain but the implementation of pure spectral technique is very difficult. We believe that spectral prediction has a potential to take a middle path between these extremes. It works in the time domain but can guess the frequency with which elements are reappearing in the stream. Furthermore, the branch predictor based on spectral prediction can provide confidence to decisions for free. This confidence can be used to allow a processor to execute both branch paths when confidence is low, and to execute only the predicted path for high confidence.

REFERENCES

1. Advanced Micro Devices. *Software Optimization Guide for AMD Athlon and AMD Opteron processors*, 2003.

2. David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, Maurice Yarrow. The NAS parallel benchmarks. *Technical Report RNR-94-007,* NASA Ames Research Center, March 1994.

3. D. Bernstein, D. Cohen, and A. Freund. Compiler techniques for data prefetching on the PowerPC. In *the International Conference on Parallel Architecture and Compilation Techniques*, p. 19, 1995.

4. Doug Burger and Todd Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.org.

5. Martin C. Carlisle, Anne Rogers, John H. Reppy and Laurie J. Hendern. Early experience with Olden. In *the Proceedings of 6th Language and Compilers for Parallel Computing*, p. 1, 1994.

6. Mark J. Charney and Anthony P. Reeves. Generalized correlation-based hardware prefetching. *Technical Report EECEG-95-1,* School of Electrical Engineering, Cornell University, February 1995.

7. Chatfield, C. (1989). *The Analysis of Time Series: An Introduction*, Fourth Edition, Chapman & Hall, New York, NY.

8. Tien-Fu Chen and Jean L. Baer. An effective on chip preloading scheme to reduce data access penalty. In *the Proceedings of the Coneference of Supercomputing,* p. 176, November 1991.

9. Tien-Fu Chen and Jean L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 51, October 1992.

10. J. Collins, D. Tullsen, H. Wang, and J. Shen. Dynamic speculative precomputation. In *34th International Symposium on Microarchitecture*, December 2001.

11. Robert Cooksey, Stephen Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 201, October 2002.

12. F. DahlGren, M. Dubois, and P. Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *the Proceedings of International Conference of Parallel Processing*, p. 56, October 1993.

13. Box, G. E. P., and Jenkins, G. (1976), *Time Series Analysis: Forecasting and Control*, Holden-Day.

14. L. Harrison and S. Mehrotra. A data prefetch mechanism for accelerating general computation. 1351. University of Illinois at Urbana-Champaign, Champaign, IL.

15. Zhingang Hu, Margaret Martonosi, and Stefanos Kaxiras. TCP: Tag Correlating Prefetchers. In *the 9th International Symposium on High Performance Computer Architecture*, p. 317, February 2003.

16. P. Glaskowsky. Pentium 4 (Partially) Previewed. *Microprocessor Report*, August 2000.

17. Intel Corporation, *Intel Pentium 4 and Intel Xeon Processor Optimization*, 2001.

18. Doug Joseph and Dirk Grunwald. Prefetching using Markov Predictors. *IEEE Transactions on Computers*, 48(2), p. 121, February 1999.

19. Norman P. Jouppi. Improving direct-mapped cache performance by the addition of the small fully associative cache and prefetch buffers. In *the 17th Annual International Symposium on Computer Architecture*, p. 364, May 1990.

20. Martin Kampe, Per Stenstrom, and Michel Dubois. The FAB Predictor: Using Fourier Analysis to Predict the Outcome of the Conditional Branches. In *the 8th International Symposium of High-Performance Computer Architecture* (HPCA'02), p. 223, February 2002.

21. Gery Kane. MIPS R2000/R3000 RISC Architecture. Prentice Hall, 1987.

22. An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-Block prediction and Dead-Block Correlating Prefetchers. In *the 28th International Symposium of Computer Architecture*, p. 144, July 2001.

23. Mikko H. Lipasti, William J. Schimidt, Steven R. Kuenel, and Robert R. Roediger. Spaid: Software prefetching in pointer and call intensive environment. In *the 28th Annual International Symposium of Computer Architecture*, p. 231, November 1995.

24. Mikko H. Lipasti, Christopher B. Wilkerson and John P. Chen. Value locality and load value prediction. *In the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

25. Chi-Keung Luk and Todd C. Mowry. Compiler based prefetching for recursive data structures. In *the 7th Inernational Conference on Architectural Support for Programming Languages and Operating Systems*, p. 222, October 1996.

26. Scott McFarling. Combining branch predictors. *Technical Report TN-36m*, Digital Western Research Laboratory, June 1993.

27. Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *the 5th International Conference of Architectural Support for Programming Languages and Operating Systems*, p .62, October 1992.

28. Kyle J. Nesbit and James E. Smith. Prefetching with a global history buffer. In *the 10th International Symposium on High Performance Computer Architecture* (HPCA'04), p. 96, February-2004.

29. Kyle J. Nesbit, Ashutosh S. Dhodapkar, and James E. Smith. AC/DC: An adaptive data cache prefetcher. In *the 13th International Conference on Parallel Architecture and Compilation Techniques*, p. 135, 2004.

30. Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as secondary cache replacement. In *the 21st Annual International Symposium on the Computer Architecture*, p. 24, April 1994.

31. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. "A case for intelligent RAM: IRAM. *IEEE Micro*. April 1997.

32. Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms. In *the 37th International Symposium on Microarchitecture*, 2004.

33. Amir Roth, Andreas Moshovos, and Gurinder S. Sohi. Dependence based prefetching for linked data structures. In *the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

34. Amir Roth and Gurinder S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001.

35. V. Sanathanam, E. H. Gornish, and C. W. Hsu. Data Prefetching on HP PA-8000. In *the 24th Annual International Symposium on the Computer Architecture*, p. 264, April 1997.

36. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 45, 2002.

37. A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computers*, p. 7, 1978.

38. The Standard Performance Evaluation Corporation. http://www.spec.org

39. K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 257, Nov. 2000.

40. Joel M. Tendler, Steve Dodson, Steve Fields, Hung Le, and Balaram Sinharoy. Power4 System Microarchitecture. *Technical white paper*, 2002.

41. D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

42. K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, April 1996.

43. C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, June 2001.