ACCURATE, TIMELY DATA PREFETCHING FOR REGULAR STREAM, LINKED DATA STRUCTURE, AND CORRELATED MISS PATTERN

By

GANG LIU

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2010

To my family

ACKNOWLEDGMENTS

I would like to gratefully and sincerely thank Dr. Jih-Kwon Peir for his guidance, understanding and patience during my graduate studies at University of Florida. He's a man with great passion and persistence in research. I’m blessed to have him as my advisor.

I would also like to thank all committee members Dr. Timothy Davis, Dr. Randy Chow, Dr. Jeffrey Ho and Dr. Liuqing Yang for their valuable comments, productive suggestions. I appreciate all the help from my colleagues, Xudong Shi, Zhen Yang, Feiqi Su, Chung-Ching Peng, Zhuo Huang, David Lin and Jianmin Chen. I am also thankful to the CISE system staffs who maintained the machines on which I ran tons of simulations.

Most importantly, none of this would have been possible without the love and patience of my family in China. My sweet family has been a constant source of love, concern, support and strength all these years. This dissertation is dedicated to them: my mom and dad, my grandparents, my brother and his family.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

ACCURATE, TIMELY DATA PREFETCHING FOR REGULAR STREAM, LINKED DATA STRUCTURE, AND CORRELATED MISS PATTERN

By

Gang Liu

December 2010

Chair: Jih-Kwon Peir
Major: Computer Engineering

Data prefetching is an efficient way to improve system performance and mitigate the Memory Wall problem caused by continuously widening performance gap between processor and memory. There are three cache miss patterns that are well suited for data prefetching: regular stream, linked data structure and correlated miss pattern. This dissertation presents accurate and timely prefetching schemes to prefetch these cache miss patterns efficiently.

A regular stream is sequence of nearby misses when their addresses follow the same ascending or descending direction in a small memory region. It is a well studied cache miss pattern, but the state-of-the-art stream prefetcher design has some shortcomings. In this dissertation, several enhancement techniques are introduced to improve the regular stream prefetcher including an integrated stride prefetcher, stream noise removal, early start of repeated short stream, and dead stream removal. The performance result demonstrates that these enhancement techniques are indeed improving the stream prefetcher performance.

A linked data structure (LDS) consists a set of nodes which are traversed through link pointers declared inside the node structure. Simple linked lists, trees, and graphs

are examples of linked data structures. LDS traversals in applications show irregular cache miss patterns, hence cannot be prefetched by stream/stride prefetchers. In addition, due to limited amount of work between visiting adjacent nodes, the LDS traversal creates a tight load-load dependent chain, often referred to as a pointer-chasing problem. Prefetching of the next node is delayed when accessing the current node encounters a cache miss. This dissertation presents a semantics-aware prefetcher for accurate and timely LDS prefetching. The semantics-aware prefetcher dynamically establishes node structure of LDS through simple experimental rules based on the LDS traversal history. This dissertation also introduces three leap prefetching techniques to improve the lateness problem existing on the LDS prefetchers.

Correlated misses are a sequence of cache misses that are repeated in the future. Correlation-based prefetchers record the miss sequences in a history table for prefetching when the recorded miss sequences reappear. To be effective, correlation-based prefetchers must record long cache miss history and incur significant storage overhead. In recent correlated prefetchers, the miss correlation history is saved in off-chip DRAM. It incurs extra delays and memory traffic to fetch the correlations for triggering prefetches. To solve this problem, we introduce a novel prefetching approach which encodes per-block miss correlations in compressed DRAM to provide unbounded miss history with minimal space and bandwidth overheads. Furthermore, the miss correlations can be fetched along with the content of the block to avoid extra off-chip accesses to obtain prefetch information.

CHAPTER 1
INTRODUCTION

With fast advances in processor technology, the speed gap has been continuously widening between processors and main memory as shown in Figure 1-1 [25]. It usually takes hundreds of processor cycles to access the off-chip memory. The execution time of a program consists two parts: processor execution time and memory stall time. With the wider performance gap, the program execution time becomes more and more depending on the memory stall time [25]. It is especially true for memory-intensive workloads, which are very popular in today's commercial and scientific workloads. This phenomenon is referred as the Memory Wall problem [70]. The Memory Wall problem becomes more severe with the emergence of the Chip MultiProcessors (CMP) where multiple processing units (or cores) are integrated in a single chip to improve the overall chip performacne.

Memory hierarchy is designed to mitigate the Memory Wall with small but fast caches. Cache takes advantage of temporal and spatial memory reference locality by storing recently used memory blocks in caches for future usage. The memory stall time can be decided by the formula below [25]:

Memory stall time = (memory accesses) * (cache miss rate) * (miss penalty)

Out-of-order processors exploit Memory-Level Parallelism (MLP) [74, 75, 45, 62, 21] to reduce memory stall time by fetching multiple memory blocks in an overlap fashion. But given the fact that the maximal number of on-fly demand misses is limited to instruction window size, as well as restricted by data dependencies, exploiting MLP has limited effectiveness in solving the memory wall problem. Many recent works [5, 40, 4, 26, 11, 46] propose to reduce memory stall time through different caching

mechanisms other than the traditional cache using Least-Recently-Used (LRU) replacement policy. The proposed caching mechanisms, however, are only effective on few workloads with special memory reference patterns. Essentially, due to limited cache capacity, the required working set of applications may not fit into the cache and causes frequent accesses to the memory. Consequently, the memory wall problem remains as the bottleneck for many applications.

Prefetching is an important mechanism to reduce memory stall time [30, 12, 24, 31, 42, 58, 60]. Based on the memory reference history, it fetches blocks speculatively in advance into caches. For correct prefetches, the prefetched blocks will be used by demand memory accesses, thus reduce cache miss rate and memory stall time. Recent works [16, 64, 65, 55, 14, 19, 56, 43, 58] in this research area show that effective prefetching mechanisms can significantly reduce cache miss rate.

Prefetching includes data prefetching and instruction prefetching. Since data misses are often dominant among all cache misses and provide more challenges for prefetching, this dissertation will focus on data prefetching. Computer industry has shown much interest on data prefetching. In 2009, Intel sponsored the 1st JILP Data Prefetching Championship (DPC-1) contest [10] to look for innovative prefetcher designs.

Along with all the benefit discussed before, however, all prefetching methods must overcome four serious challenges: accuracy, timeliness, coverage, and storage overhead. Inaccurate prefetches that bring blocks into the cache without using them increases the memory bandwidth requirement, consumes more energy, and pollutes the caches. Timely prefetches bring in the instruction and data before demand usages to

avoid cache misses. The overall miss coverage is measured by the ratio of miss reduction based on the misses without prefetching. Such a ratio is a result of the miss reduction from prefetches and the miss increase due to cache pollution. A refined prefetch taxonomy is given in [59]. Last but not the least, any practical prefetching method must constrain its storage overhead. Such overhead can be exacerbated in many-core chip-multiprocessors (CMPs) for recording the miss history from multi-thread or multi-program applications.

In this dissertation, three cache miss patterns are studied for data prefetching. We will investigate the behaviors of Regular Streams, Linked Data Structures (LDS), and Correlated Misses (Figure 1-2) and present solutions to accurately and timely prefetch these missing patterns. We will also evaluate hardware cost in establishing the history of cache misses for data prefetching.

1.1 Cache Miss Patterns

1.1.1 Regular Stream

Many cache miss patterns often present regularity in nearby missing addresses when the addresses follow the same ascending or descending direction in a small memory region [58, 60]. To demonstrate regular miss stream patterns, we collected a snapshot of 50000 consecutive L2 misses of mcf from SPEC2000 benchmark suite on a 1MB cache. We can observe in Figure 1-3 that regular stream patterns exist at the beginning and the ending sections of the simulation. We can also see a long period with irregular missing addresses, which we will explore in next subsection.

Accesses to a regular stream can be consecutive, with a constant stride, or non-stride, but keeps in the same direction. Such regular streams can be prefetched by

following its access direction. Due to its simplicity and effectiveness, the stream-based prefetching schemes have been implemented in commercial processors [58, 60].

1.1.2 Linked Data Structure

Besides regular streams, applications often manifest a mixture of regular and irregular cache miss patterns [36, 19]. Figure 1-3 illustrates such a situation. After tracing the mcf example, it shows that these irregular cache misses occur in function Refresh_Potential when the program traverses through a tree-like Linked Data Structure (LDS) [35, 36, 49, 15, 19]. In this example, one of the links in each node is selected for traversal to the next nodes. Such LDS traversal patterns are common in many applications and are also referred to as a pointer-chasing problem [36, 19]. During traversal, although the address of the next node is included in the current node, it is not available until the current node is processed. There are often very few instructions between visiting two adjacent nodes, hence encounter a tight load-load dependences. Timely prefetches for the LDS traversal become a big challenge unless the history of miss address can be recorded in a history table.

1.1.3 Correlated Misses

Cache misses often exhibit streaming behavior, i.e. a sequence of cache misses has a high tendency of being repeated in the future as reported in [13, 65, 55, 14, 66, 56, 67]. The sequence of missing addresses can be irregular and may not have the semantics as LDS traversal, thus brings more challenges in prefetching the irregular but repeated miss patterns. Figure 1-4 shows an example of correlated misses, where A, B, C, D, E, F, G are consecutive misses and repeat in two consecutive iterations. Between iterations, there are a large number of other cache misses so that A, B, C, D, E, F, G are replaced from the cache before the next iteration.

To prefetch the correlated misses, correlation-based prefetchers [9, 29, 13, 65, 55, 56] record the miss sequences in a history table for prefetching when the recorded miss sequences reappear. To be effective, correlation-based prefetchers must record long cache miss history and incur significant storage overhead, especially for memory-intensive workloads. Saving the miss history in off-chip DRAM is a practical implementation, but incurs access latency and consumes memory bandwidth which can lead to performance degradation.

Although the three cache miss patterns have totally different characteristics, they also have overlaps with one another as shown in Figure 1-2. For example, LDS miss patterns can sometimes show behavior of regular streams, which is often due to the initial memory allocation of LDS structures in consecutive address space. Also, the LDS traversal in Figure 1-3 repeats for many times, thus also qualifies as correlated misses. Due to different characteristics of the three cache miss patterns, it will be beneficial to incorporate multiple prefetchers to handle different missing patterns in future processors.

1.2 Dissertation Overview

For the three cache miss patterns introduced in Section 1.1, i.e., Regular Stream, LDS and Correlated Misses, three corresponding works are presented to prefetch these cache miss patterns accurately and timely. The first work is improving an existing stream prefetcher with some enhancement techniques. The second work prefetches LDS patterns by dynamically establishing semantics of the node structure, and introducing several leap techniques to help timely prefetching. The third work handles correlated misses by encoding per-block miss correlations in compressed DRAM for data prefetching.

1.2.1 Enhanced Stream Prefetcher

In the first work, we present the enhanced stream prefetcher which competed in the 1st JILP Data Prefetching Championship (DPC-1) contest [10]. We describe several enhancement techniques to improve the state-of-the-art stream prefetcher. First, the enhanced stream prefetcher takes streams with long stride into consideration to avoid wasteful prefetches. Second, accessing a node in tree or graph structures may have a different direction than the traversal direction through the structure. The enhanced stream prefetcher eliminates this type of noise for establishing the stream. Third, regular streams for array accesses are often repeated. Initiating penalty can be avoided by early re-establishing a repeated stream. Fourth, an established stream may be dead before being removed from the stream prefetching table. A dead stream removal scheme reduces inaccurate prefetches. Performance evaluations based on a set of SPEC2000 and SPEC2006 benchmarks show that the Enhanced Stream Prefetcher makes significant improvement over the original stream prefetcher.

1.2.2 Semantics-Aware Prefetching of Linked Data Structure

In the second work, we describe a Semantics-Aware hardware solution to improve the lateness issue in LDS prefetching. We observe three characteristics in LDS traversal that enable leap prefetching to fetch the nodes further ahead in the traversal path. First, in the traversal of a tree-like LDS, each node can be visited multiple times using a different pair of links in and out of the node in each visit. Instead of prefetching through a single link, prefetches on multiple links can fetch additional nodes for future visits, which is similar to the software-based greedy prefetcher [36]. Second, a logical LDS node may not be aligned with the physical cache block. As a result, opportunities exist for leap prefetching from the pointer links of other logical nodes located in the

same physical block. Third, some applications establish a pointer array to linked lists as observed in [32]. The traversal sequence of the pointer array often exhibits regular stream or stride behavior. With an integrated stream/stride prefetcher, leap prefetching on future visiting lists of nodes is possible when the pointer array is prefetched.
In order to trigger the leaped LDS prefetching accurately, however, it is essential to capture the node semantic structure of the pointer links dynamically. In this work, we present simple experimental rules in detecting pointer links based on the LDS traversal history. When traversing between adjacent nodes, the same or different links must be used. The experimental rules detect the load-load dependence between the adjacent traversal links. When a missing block due to LDS traversal becomes available, prefetches based on the semantic structure of the traversal node can be issued. Performance evaluations based on several LDS-intensive applications have demonstrated the effectiveness of the Semantics-Aware prefetcher.

1.2.3 Encoding Per-Block Miss Correlations in Compressed DRAM for Data Prefetching

To prefetch the correlated misses, correlation-based prefetchers [9, 29, 13, 65, 55, 56] record the miss sequences in a correlation table for prefetching. In the third work, we present a novel prefetching approach to encode per-block miss correlations in compressed DRAM to provide unbounded miss history with minimal space and bandwidth overheads. The per-block miss correlations are collected dynamically and encoded in each block of DRAM. Such miss correlations can be fetched automatically along with the content of the block when a miss occurs to trigger prefetches. Encoding such per-block miss correlations becomes feasible with recent development in data compression techniques [68, 76, 61, 71, 20, 72, 2, 3, 47].

In contrast to existing memory compressions for packing more blocks in DRAM with variable block sizes, our approach is to simply compress individual memory blocks to free up space to record the per-block miss correlations with a constant block size. We simulate a compression algorithm called Single-Value Dynamic Encoding (SVDE) [47]. The result shows for most of SPEC benchmarks [7], in average each compressed block can have 15 extra bits, which can be used for storage of per-block miss correlations.

Although per-block miss correlations can provide unbounded history, it is constrained by the limited free bits available to encode the correlations. Given f free bits after compression for each block in DRAM, the per-block miss correlation must be packed into the available f bits. Two forms of the miss correlation are considered. First, a straight-forward method is to encode the block distance from the base block to the correlated miss block. Second, to capture spatial correlation behavior, a bitmap vector can be used to encode up to f correlated miss blocks that are located within the same spatial region, where each bit represents a correlated neighbor. Given f available bits, the bitmap correlation can cover a region of f consecutive blocks around the base block while the distance correlation records a single correlated missing block located within the region of $\pm 2^{f-1}$ blocks from the base block. Results show that the Per-Block Miss Correlations prefetcher (PBMC) indeed cover very high percentages of misses and improve performance significantly.

1.3 Dissertation Contribution

In this dissertation, we design three efficient data prefetchers for accurate and timely prefetch three different cache miss patterns: regular stream, linked data structure, and correlated miss pattern. We develop cycle-accurate simulation models including

single core or CMP along with caches, system buses, and block-interleaved multiple-banked DRAMs. These models are integrated with a full-system execution driven simulator Virtutech Simics [37] and an x86 out-of-order processor timing-model FeS2[23, 38]. These models are designed specifically for studies of data prefetching mechanisms. We collect detailed behaviors of the studied benchmarks in various ways, which can be source code, assembly code, snapshot of miss addresses or abstract example. With respect to the three prefetching schemes, this dissertation makes the following contributions.

1. Enhanced Stream Prefetcher

- We observe several shortcomings in existing stream prefetcher that will result in inaccurate prefetches and waste of memory bandwidth.
- We improve the existing stream prefetcher with several enhancement techniques and improve the performance in prefetching regular streams.
- We develop a stream prefetcher with the enhancement techniques, as well as GHB-Distance prefetcher [43] for comparison.

2. Semantics-Aware LDS prefetcher

- We address the main problem of lateness issue in the previous LDS prefetcher designs, and the potential performance improvement if LDS prefetches are issued in time.
- We give our solution of LDS prefetching which can alleviate the lateness issue. Our solution is pure hardware solution which can dynamically identify LDS node structures. We introduce several leap prefetching techniques that can trigger prefetches early.
- We develop a Semantics-Aware LDS prefetcher, as well as dependence-based LDS prefetcher (DBP) [49] and content-directed prefetcher (CDP) [16] for comparison.

3. Per-Block Miss Correlation Prefetcher

- We address the problem with previous correlation-based prefetchers is either huge on-die correlation table or extra metadata traffic to DRAM.

- We propose our solution to store the per-block miss correlations in compressed DRAM. It is proved to require neither on-die correlation table nor extra metadata traffic.
- We develop the PBMC prefetcher, and the temporal memory streaming and spatial memory streaming prefetchers.

1.4 Performance Evaluation Methodology and Benchmarks

To demonstrate the advantages of the enhanced stream prefetcher, we selected twelve benchmarks with high L2 Misses-Per-Kilo-Instructions (MPKI) from SPEC2000 and SPEC2006. Trace-driven simulations were carried out using the CMPsim tool set provided by the 1st JILP Data Prefetching Championship competition committee [10]. The traces were collected from each benchmark by fast-forwarding 40 billion instructions, and then collected traces for the next 100 million instructions.

The simulation framework models an out-of-order core with the basic parameters as outlined in Table 1-1. Two L2 cache sizes and two memory bandwidths are considered resulting in three L2 cache configurations as requested by the competition committee. For the other two works: Semantics-Aware LDS prefetcher and PBMC prefetcher, we use a full-system execution-driven simulator Virtutech Simics [37] with an integrated cycle-accurate processor timing-model FeS2 [23, 38] in the study. FeS2 leverages the x86 decoder and functional u-op implementations from PTLsim [73] and simulates a 15-stage out-of-order x86 processor. The microarchitecture configuration follows the default setting in the simulator. A cycle-accurate cache and memory model is developed for evaluating various L2 prefetching methods. Multiple-channel memory controller connects the L2 cache with multiple-banked DRAMs for handling regular miss and L2 prefetch requests. In addition to regular L2 MSHRs, a prefetch queue is added to record in-flight prefetch requests. In addition, a writeback queue is used in handling

the updates of the prefetch metadata. The demand L2 misses have higher priority over the prefetch requests in competing for the system bus and the DRAM ports. Table 1-2 summarizes the basic system configuration and parameters used in our simulation.

In the work of Semantics-Aware LDS prefetcher, twelve LDS-intensive benchmarks are selected for evaluations of various prefetching methods similar to the previous study [19]. The selected benchmarks are: gcc06, mcf06, omnetpp06, perl06 and xalanc06 from SPEC2006, ammp, mcf00, parser and vpr from SPEC2000, and bisort, health, and mst from Olden [48]. All benchmarks are simulated for 200 million instructions for collecting the statistics. For SPEC2000 and SPEC2006 benchmarks, we use the ref as the input. For SPEC2000, we bypass certain instructions based on the study in [52]. For SPEC2006, the starting check point was captured from the region with high cache miss frequency. For Olden, we use the input size used in [36], but long enough to run for 200 million instructions, and start simulations after the program initialization region.

In the work of PBMC prefetcher, data-parallel workloads are used to evaluate the prefetchers for many-core architecture. One common characteristic is that these workloads have plenty of data parallelism that many-core architecture can explore to improve performance (in terms of response time or throughput) [34]. To explore data parallelism, two common software models are used. Multithreading is one model where a single dataset is processed in parallel by multiple threads to speed up the overall response time. OpenMP and posix-thread are two common frameworks used to implement multithreading workloads. In our study, equake and swim from SPECOMP [57] as well as stream [39] are examples of OpenMP workloads. lbm [44], radix-sort

[51], facesim [8], and ocean [69] are examples of workloads parallelized using posix-threads. lbm is the parallel version of the one in SPEC. radix is a sorting application using radix-sort algorithm. facesim simulates facial muscles on a tetrahedron mesh using the Netwon-Raphson method. Single-program-multiple-data (SPMD) is the common model to explore data parallelism where a single program is executed on multiple data simultaneously to improve throughput. The map part of the well-known map-reduce model is an example of SPMD. We approximate this type of workloads by running multiple copies of the single-threaded workload on an 8-core system. In our study, we run eight copies of libquantum [57] and eight copies of health [6], which were selected due to their high MPKI and irregular miss patterns. In addition, we study hybrid workloads where it consists of two workloads with different behaviors. For this, we run four copies of sphinx and four copies of health together (denoted as sph_hea) where sphinx is dominated by regular stream/stride misses.

1.5 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 describes the first piece of the dissertation, an enhanced stream prefetcher, which improves the performance of existing design with several enhancement techniques. Chapter 3 presents a semantics-aware LDS prefetcher. The semantics-aware dynamically LDS node structures for the purpose of prefetching. Three leap prefetch techniques are employed to alleviate the lateness issue in LDS prefetching. Chapter 4 demonstrates an innovative correlation-based prefetcher based on per-block miss correlations which are stored in compressed DRAM. Chapter 5 concludes the dissertation.

Table 1-1. Simulator Configuration

Configuration Item	Value
Issue width	4
Instruction Window	128 entries
L1 cache	32KB, 8-way,I/D caches, 1 cycle
L2 cache	512KB/2MB, 16-way, 20 cycles
Memory latency	200 cycles
Configuration 1 (*c1*)	2MB L2, 1000 requests/cycle
Configuration 2 (*c2*)	2MB L2, 1 request/10 cycles
Configuration 3 (*c3*)	512KB L2, 1 request/10 cycles

Table 1-2. Simulation parameters

Processor & Memory Hierarchy
4GHz, 15-stage out-of-order x86 core
Fetch/rename/execution/retire width: 3/4/4/4
ROB size: 80, Physical registers: 80, LQ/SQ entry: 64
Hybrid branch predictor: g-share, 4K-entry BTB, 32-entry return address stack
Memory Hierarchy
L1 I/D Caches: 32KB 4-way, 64B line, 2 read 1 write ports, 1-cycle latency, remote L1: 5-cycle
Shared L2 Cache: 1MB (4MB for 8-core CMP), 8-way, 64B line, 2 r/w ports, 20-cycle latency
Coherency protocol: MOESI
L1/L2 MSHRs: 32/32
Prefetch Queue: 96
Write-back Queue: 64
Memory: 200-cycle DRAM latency, 64 block-interleaved DRAM banks
Memory Bandwidth : 3 channels, 1.33GHz, bi-direction buses, 8-byte width, 32GB/sec

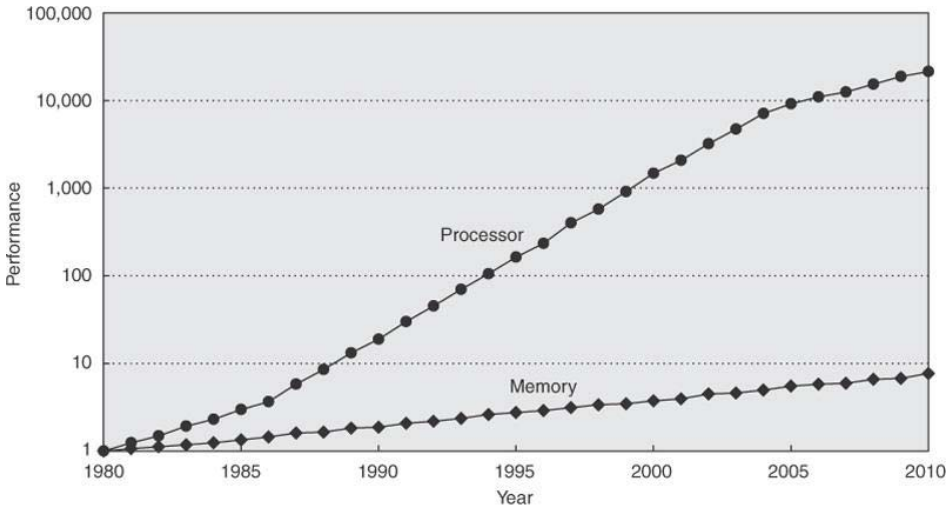

Figure 1-1. Performance comparison of processor and memory for the last 30 years.

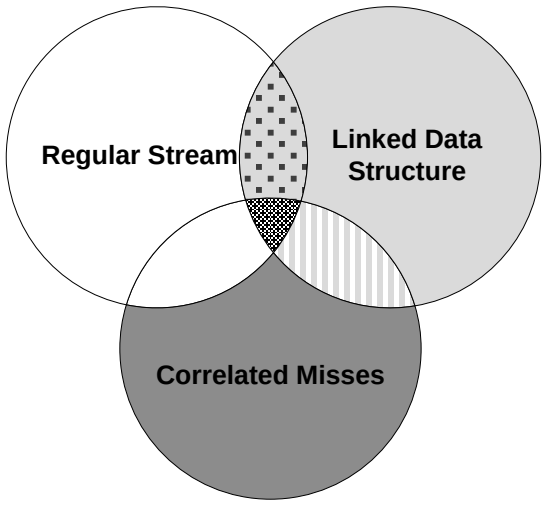

Figure 1-2. Three cache miss patterns

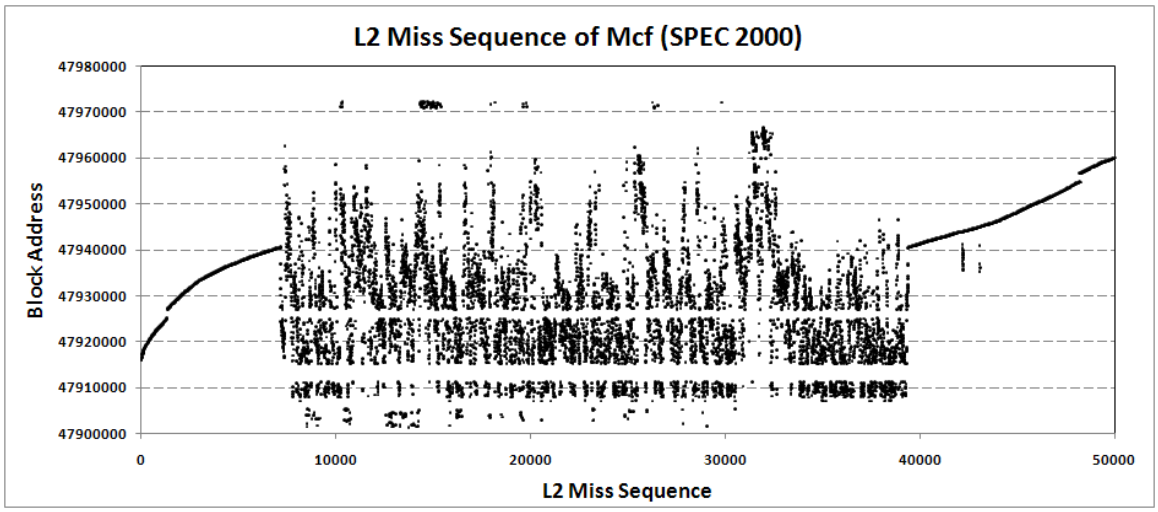

Figure 1-3. Snapshot of L2 miss sequence from SPEC2000 benchmark mcf

Iteration 1: **A – B – C – D – E – F – G**

……

Iteration 2: **A – B – C – D – E – F – G**

Figure 1-4. Abstract example of correlated misses

CHAPTER 2
ENHANCEMENTS FOR ACCURATE AND TIMELY STREAM PREFETCHING

2.1 Introduction

There are several existing prefetchers to handle regular stream miss patterns [30, 12, 24, 31, 42, 58, 60]. Among these prefetchers, the stream prefetcher captures a sequence of nearby misses when their addresses follow the same positive or negative direction in a small memory region. Once a stream is identified, a demand miss targeting the current active stream triggers prefetches of consecutive blocks in the detected direction. Due to its simplicity and effectiveness, the stream prefetcher has been implemented in commercial processors [58, 60].

In this chapter, we present several enhancement techniques for optimizing a stream prefetcher. First, the stream prefetcher is augmented with a stride distance. Upon detecting memory accesses with a constant stride (measured in bytes) of longer than one block, the stream prefetcher prefetches blocks according to the detected stride. Second, we observe that in a few applications, accessing a data structure through pointers such as trees and graphs, each node in the data structure may occupy more than one block. Hence, accesses within a node may have a different addressing direction than the traversal direction through the nodes. When such a case is detected, we allow a stream to be formed by ignoring the noise, i.e. an adjacent block access in the opposite direction. Third, we observed that regular streams for array accesses are often repeated. To reduce the penalty of re-initiating a stream, we allow a previous repeated stream to launch again after the old stream is caught up by a new stream. Fourth, a short stream may be dead before being replaced from the stream table. Subsequent hits to a dead stream initiates inaccurate prefetching without going through

the needed training stage. Detecting and removing aging short streams from the stream table leads to more accurate prefetching.

The remainder of this chapter is organized as follows. Section 2.2 provides the basic design of stream prefetcher. Section 2.3 describes the benefit of constant stride, stream repetition, as well as the challenge in handling stream noise and dead stream removal. Section 2.4 describes the details of the enhancement techniques. Section 2.5 presents the evaluation results. Related work is given in Section 2.6 followed by a brief summary in Section 2.7.

2.2 Stream Prefetcher Basics

The stream prefetcher we model is based on the one presented in [58] which is originated from the IBM POWER4 processor [60]. The stream prefetcher uses a stream table to keep track of multiple access streams. All established streams (also referred as trained streams) in the stream table are monitored against the cache misses. When an incoming memory request falls into the current monitor window of a trained stream, the stream prefetcher prefetches consecutive blocks according to the direction of the trained stream.

A demand missing block address enters the stream table in an untrained state if the missing block has not been recorded. An untrained stream becomes trained based on the following conditions. First, the next two consecutive misses located in the same memory region as an untrained miss are examined. In the design reported in [58], the memory region covers 16 blocks before and after the original missing block. Second, these three consecutive misses are in the same ascending or descending direction starting from the original miss.

The aggressiveness of a stream prefetcher is controlled by the distance and the degree of the stream as illustrated in Figure 2-1. The original miss address and the current stream window defined by the start and the end block address of each trained stream are recorded in the stream table. The number of blocks from the start to the end blocks determines the prefetch distance, which indicates the stream monitor region as well as controls how far ahead of the demand access stream that the prefetcher can prefetch. The prefetch degree, on the other hand, controls the number of consecutive blocks for each stream prefetcher. When a new memory request falls into the current monitored region of a trained stream, the stream prefetcher prefetches the next n consecutive blocks starting from the end of the monitored region, where n is the prefetch degree. Afterwards, the monitored start and end address are advanced by n blocks to keep the stream moving along the stream direction.

2.3 Stream Enhancement Techniques

2.3.1 Constant Stride Optimization

The first enhancement is to integrate long-stride prefetching into the stream prefetcher. In the original stream prefetcher, streams are prefetched by consecutive blocks. In real applications, it is not uncommon that memory accesses are followed a constant stride across multiple blocks. Although such long-stride accessing patterns can be captured by a stream prefetcher, stream prefetching of consecutive blocks wastes memory bandwidth and pollutes the caches. For example, in examining scanner.c in art from SPEC2000 Benchmark (Figure 2-2), we can identify a constant-stride access pattern across multiple blocks. Note that the memory references in this routine causes 80% of all misses in art on the baseline 1MB L2 cache without prefetching. As shown in the memory allocation part, the size of each element of the two arrays bus and tds is 88

bytes. Each element of two arrays are allocated one-by-one in a round-robin fashion. Therefore, two adjacent elements in bus are 192 bytes apart after padding the arrays with 16 bytes. Given a cache line size of 64 bytes, consecutive accesses to array bus span across 3 blocks since &bus[i+1][j] - &bus[i][j] = 192 bytes. As a result, 2 out of 3 prefetched blocks are wasted by the original stream prefetcher.

In the enhanced stream prefetcher, a straight-forward solution dynamically detects the stride distance information. If a constant stride is detected, instead of prefetching consecutive blocks, the constant stride is used to calculate the correct blocks to avoid extra prefetches. More design details will be given in the next section.

2.3.2 Noise Removal

In training a stream in a stream prefetcher [58, 60], three consecutive misses addressing in a small memory region are examined. These three misses must follow the same ascending or descending direction to successfully train the stream. When a training stream fails, it is likely that two consecutive misses are addressing memory blocks from the opposite directions with respect to the first miss. After examining the unsuccessfully trained streams in soplex, it is interesting to see that out of 11484 unsuccessfully trained streams, 7343 are due to an access to the next adjacent block in the positive direction (i.e. a positive distance of one block). We traced the memory reference pattern of soplex, and found it behaves as illustrated in Figure 2-3. Although the overall direction of the stream is descending, the positive one-block jumps keep preventing the stream from being trained. To remedy this problem, we allow a training stream to stay untrained when the subsequent miss occurs in the opposite direction from the current miss and one of the misses in the opposite direction is accessing the

adjacent block. In other words, the adjacent block access is considered as noise and is ignored in training the stream as illustrated in Figure 2-3.

2.3.3 Early Launch of Repeat Stream

Besides the constant stride accesses in scanner.c (Figure 2-2), we also observe that a stream access is often repeated. In the nested loop of the example, the streaming array bus is accessed repeatedly 11 times with exactly the same start and end addresses. In addition, the repetition of the streaming accesses is separated only by a few instructions. It is beneficial to initiate the stream prefetching again from the recorded original stream address when the current stream is coming to an end. By keeping previous long streams in the stream table, we can detect repeated streams once two streams overlap in the monitored region. Early launching a repeated stream can reduce the penalty of initiating a new stream prefetching.

2.3.4 Dead Steam Removal

Given the relatively loose condition in training a stream as described in Section 2.2, many streams can be established even if they are not an accurate stream for prefetching. Furthermore, we also observe that for many short streams that remain inactive for a long period of time, the stream likely has come to an end. These dead streams may still stay in the stream table and can accidentally catch an incoming memory access to trigger prefetches. These incorrect prefetches pollute the cache and waste memory bandwidth.

The number of dead streams in the stream table goes up with the table size. Simulation results show that with a 128-entry stream table, roughly 88% and 62% of the trained streams have stayed in the table longer than 100K cycles without triggering any prefetches respectively for art and ammp. When a prefetched block is triggered by an

old stream that has not triggered any prefetch over 100K cycles, it is useless 92% of the time. Although a smaller stream table can naturally replace streams before they die, smaller tables may suffer insufficient space to keep all the active streams. Hence, by removing dead streams dynamically based on their ages in a reasonable-size stream table, the active streams can likely be maintained without holding many dead streams and causing incorrect prefetches.

2.4 Enhanced Stream Prefetcher Design

In this section, we describe the detailed designs and operations of integrating the four enhancement techniques into the original stream prefetcher. The same two steps: training and prefetching are performed in the enhanced stream prefetcher. In the training stage, the noise removal technique is integrated to screen accesses with the noise behavior as shown in Figure 2-3. In the prefetching stage, correct memory blocks with constant-stride accesses are detected and prefetched. In addition, repeated streams are captured and triggered earlier when the current stream catches up an existing stream in the stream table. Furthermore, dead streams are removed from the stream table based on the stream age.

Figure 2-4 shows the flowchart of the enhanced stream prefetching. The basic designs and data structures are given in the following subsections. Note that for simplicity, we use separate training and streaming tables in the respective stages. They can be combined into a unified table.

2.4.1 Stream Training

Each entry in training table is depicted in Figure 2-5. Three consecutive misses in a small training window need to be captured for stream training. All the misses are represented by cache block addresses, with 26 bits for cache block address of the 1st

miss, and 5 bits representing a distance of +/- 16 blocks from the 1st miss to the 2nd miss or the 3rd miss. Hence the training window of each stream has 32 blocks. Direction is an indicator of an ascending or descending stream. The Noise Flag marks the activation of the noise removal in training. The training stage follows several steps.

1. When a cache miss occurs, both the training and the stream tables are searched. If the miss falls into the monitored region of a trained stream, the miss will not be trained again.
2. If the miss block does not fall into any training window (i.e. within positive and negative 16 blocks of the recorded miss) in the training table, a new entry is created to record the new miss for training and the LRU entry is replaced.
3. If the miss is within a training window of a recorded miss, three actions are followed.
    a. Record the block distance from the 1st miss in case the miss is the 2nd miss of the training stream.
    b. If the 3rd miss is detected and all three misses are following the same direction, the stream is trained and is moved to the stream table for prefetching.
    c. If the three misses are not in the same direction, there are two conditions.
        1) If an access to the adjacent block is detected which is in the opposite direction of the stream training, such an access is treated as a noise and removed.
        2) If the three misses do not satisfy the noise removal condition, the 2nd miss and the 3rd miss replace the 1st miss and the 2nd miss in the corresponding training stream for continuing the training.

2.4.2 Stream Prefetching

After a stream is successfully trained, the stream is moved from training table to stream table. The information recorded in each stream table entry is given in Figure 2-6. Orig Addr is the cache block address of the 1st miss from the training table. Start Addr and End Addr form the monitored region, with 32-bit full address for End Addr, and 9 bits of block distance from Start Addr to End Addr. Last Addr records the actual last

memory access in form of byte distance from End Addr for the purpose of detecting constant stride accesses. Direction is the indicator of an ascending or descending stream. Stride records the last stride distance in bytes. StrEn flag enables a stride prefetching based on the detected stride distance. Repeat flag is used to mark a repeated stream. Finally, TimeStamp stores the age in CPU cycles of the last stream prefetching triggered by the recorded stream. The enhanced stream prefetching follows several steps.

- When a memory access falls into the monitored region of a trained stream, stream prefetching of subsequent n blocks is triggered where n is the prefetch degree. The prefetching starts from the block following the End_Addr according to the stream direction for n consecutive blocks.

- A stream with constant-stride over the length of one block can be detected dynamically and used for accurate stride prefetching. When a memory access occurs in the monitored region of a trained stream, the memory address is saved in Last Addr, and the access stride in byte granularity is recalculated. In case the new Stride matches the previous Stride, the StrEn flag is turned on and the subsequent prefetches will be based on the recorded Stride. Note that whenever a new Stride mismatches the recorded Stride, the StrEn flag is turned off and the prefetcher is reset to the stream prefetching of consecutive blocks.

- The detection and early prefetching of repeated streams work as follows. When the forwarded monitored region of an active stream overlaps with the monitored region of another inactive stream in the stream table, a repeated stream is discovered under two conditions. First, the two streams have their starting addresses closely to each other, i.e., the two streams start from nearby addresses. Second, both streams are long streams which cover more than 256 blocks. Upon discovery of such a repeated stream, prefetching of the inactive stream is triggered from its original address.

- The age of an existing stream is used to identify and remove potential dead streams from the stream table, where the age is measured from the last time when a stream prefetching is triggered. A global TimeStamp is used to calculate the stream age. The current TimeStamp is saved into the stream table whenever a memory access occurs to a stream. The age of all the streams in the stream table are checked periodically. The potential dead stream is identified and removed when the stream has been idled for a long period time (> 10K cycles) and the stream is a short stream covering less than 256 blocks.

2.5 Experimental Results

2.5.1 Performance of Enhanced-Stream Prefetcher

We evaluate and compare three prefetch schemes, including the PC-based Distance prefetcher using a Global History Buffer (GHB-Distance) [43], the original Stream prefetcher (Stream) [58] and the Enhanced Stream prefetcher (Enhanced-Stream). All prefetchers prefetch memory blocks directly into the L2 cache. The simulation methodology is described in Section 1.4, parameters for the simulator are listed in Table 1 1. The simulated table sizes for the three prefetchers are given in Table 2-1. Both the prefetch width and depth for GHB-distance are 16 and the prefetch degree and distance are 4 and 64, respectively, for both stream-based prefetchers. Under the allowed space budget, we simulate multiple table sizes for maintaining the stream history and selected the size that demonstrates the highest performance for both stream-based prefetchers. For achieving the best performance, the results show that both stream prefetchers require very little history information.

Figure 2-7 shows the normalized CPI comparison of the three prefetching schemes where the CPIs are normalized to the baseline design without any prefetching. In this figure, the twelve selected benchmarks are sorted from left to right in the descending order of the MPKI. We can make several important observations. First, on the arithmetic average of all workloads, the performance improvements over the base CPI are 27%, 37%, and 38% for GHB-Distance, Stream, and Enhanced-Stream prefetchers for the c1 configuration, 16%, 29%, and 42% for the c2 configuration, and 26%, 44%, and 55% for the c3 configuration, respectively. Overall, Enhanced-Stream outperforms GHB-Distance and Stream respectively by about 30% and 14%.

Second, among the four enhancement techniques, stride prefetching gains the most benefit. For art and mcf, the performance gains from stride prefetching are about 45% and 28%, respectively, revealing that stream prefetching of consecutive blocks is wasteful and inaccurate in these applications. Third, different benchmarks show very different results with respect to the three prefetching schemes. Enhanced-Stream is most effective for art and mcf which have the highest MPKI and most beneficial from stride prefetching. Early prefetching of repeated stream works well for art with about 6% improvement on 512KB L2 cache. The noise removal scheme shows some impacts on soplex for a minor performance gain about 1%. The dead stream removal scheme is very effective in many applications when large stream tables are used. We will show the sensitivity study results in Section 2.5.2. Fourth, GHB-Distance performs worse than the other two schemes for most applications. However, it shows slight edge over stream-based prefetchers on ammp and omnetpp. This is due to the long and constant distances are covered in the GHB-Distance prefetcher. Finally, we also observe that integrations of multiple enhancement techniques may cause performance interferences among one another. The results in Figure 2-7 are simulated with the combination of all four enhancement techniques.

2.5.2 Sensitivity Study

The sizes of the training table and the stream table impact the overall prefetch performance. In Figure 2-8, we collect the average CPI simulated with the three cache configurations listed in Table 1 1, and compare different combinations of various table sizes in Enhanced-Stream. For the stream table, we simulate six table sizes with 4, 8, 16, 32, 64, and 128 entries. For each stream table size, we simulate three associated

training table sizes with the number of entries equal to the same, double, and quadruple of the stream table size.

The results are plotted in Figure 2-8, where the notation n/m1, m2, m3 represents the size of the stream table (n) and three associated training table sizes (m1, m2, m3). It is interesting to observe that the stream table of 8 entries has the lowest overall CPI indicating the number of active streams is very small in all applications. Increasing the stream table size beyond 8 degrades the performance. This is due to the fact that many inactive streams are kept in the stream table and cause inaccurate prefetching. When the number of the stream table entries reduce to 4, the insufficient space to hold all active streams reduce the overall improvement. We can also observe that the performance improvement is rather insensitive to the training table size.

We also evaluate the effectiveness of dead stream removal with different stream table sizes as shown in Figure 2-9. The left figure shows the average performance with the three cache configurations for all the workloads, with three dead stream removal techniques: no removal, removal streams with the age of 100K cycles, and removal streams with the age of 10K cycles. As can be observed, when stream table size is larger than 8, dead stream removal is effective in reducing the damage of inaccurate dead-stream prefetching. However, when the stream table size reduces to 8 or smaller, dead stream removal has very little impact. This is due to the fact that small table sizes can naturally replace dead streams dynamically.

Optimal stream table size is not always the same for individual workloads. The right figure in Figure 2-9 demonstrates that optimal stream table size varies from 4 to 64 for individual workloads. For example, with 16 stream entries, swim can improve 5%

over 8 stream entries. Hence in practice, it would be a better choice to have a relatively large stream table integrated with an effective dead stream removal scheme.

2.6 Related Work

Traditional hardware-oriented sequential [30], stride [12, 24], distance [31, 42] and regular stream [58, 60] based prefetchers work well for applications with regular cache miss patterns. These prefetchers dynamically capture the regularity of a sequence of missing block addresses to speculatively prefetch the subsequent blocks. Among them, the stream prefetcher [58, 60] has been adapted in commercial processors due to its simplicity and effectiveness. Stream prefetchers prefetch consecutive blocks according to the streaming access direction when consecutive misses within a small memory region follow the same direction. The prefetching stream continues as long as subsequent memory requests fall in the monitored region of the active streams.

2.7 Summary

We present several enhancement techniques to improve the existing stream prefetcher [58, 60]. Evaluations show that the enhanced stream prefetcher can prefetch regular memory access streams more accurately and timely, and improves performance significantly based on the simulation model and workloads provided by the prefetch competition committee [10]. We also show that the space overhead of implementing an enhanced stream prefetcher is very small.

Table 2-1. Prefetcher Configurations

Prefetcher	Table configuration	Size
GHB-distance	256 IT entries, 256 GHB entries	4KB
Stream	16 combined entries	128B
Enhanced-Stream	8 training entries, 8 stream entries	256B

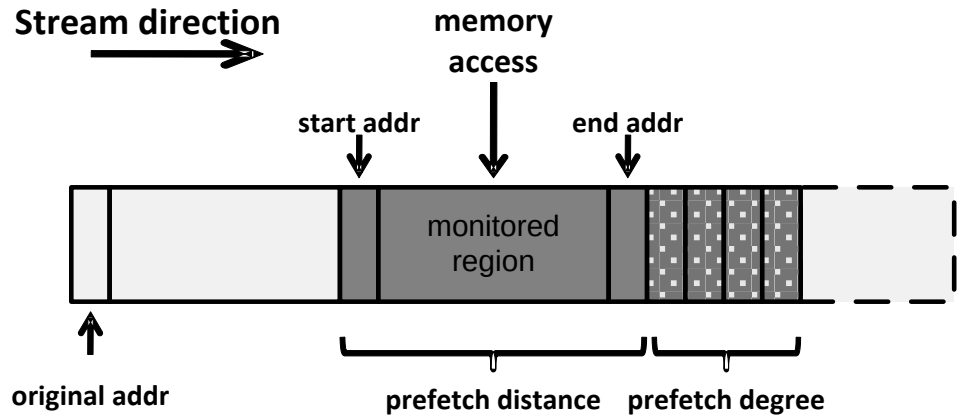

Figure 2-1. Basic design of Stream Prefetcher

Memory Allocation:

```
for (i=0;i<numf1s;i++) {
        //numf1s = 10000,numf2s = 11
        bus[i] = (double *)malloc(numf2s*sizeof(double));
        tds[i] = (double *)malloc(numf2s*sizeof(double));
}
```

Memory Access:

```
for (tj=0;tj<numf2s;tj++) {
        …
        for (ti=0;ti<numf1s;ti++)
                Y[tj].y += f1_layer[ti].P * bus[ti][tj];
}
```

Figure 2-2. Code segment from scanner.c in SPEC2000 Benchmark art

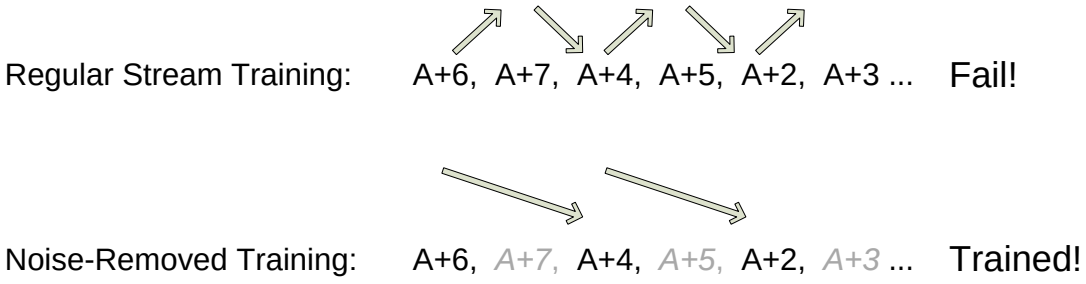

Figure 2-3. An abstract example of stream training with noise removal

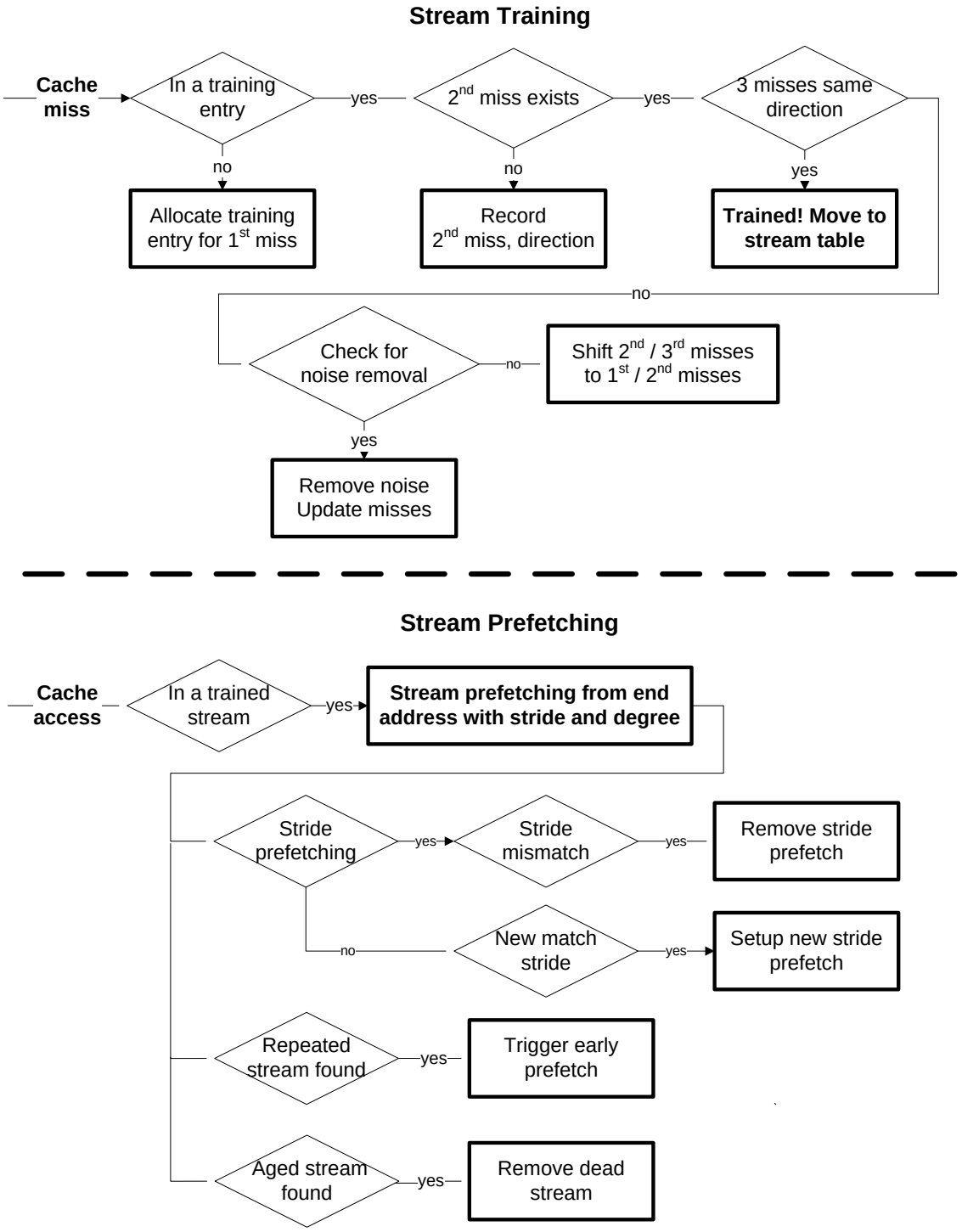

Figure 2-4. Flowchart of enhanced stream training and prefetching

1st Miss	2nd Miss	3rd Miss	Direction	Noise Flag
26 bits	5 bits	5 bits	1 bit	1 bit

Figure 2-5. Training table entry

Orig Addr	Start Addr	End Addr	Last Addr	Direction	Stride	StrEn	Repeat	TimeStamp
26 bits	9 bits	32 bits	16 bit	1 bit	9 bits	1 bit	1 bit	32 bit

Figure 2-6. Stream table entry

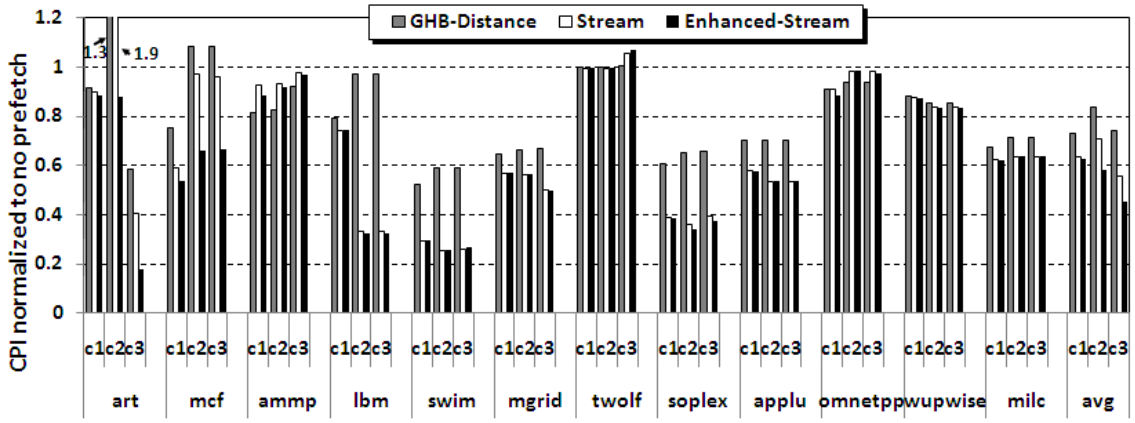

Figure 2-7. CPI comparisons for the three prefetching schemes

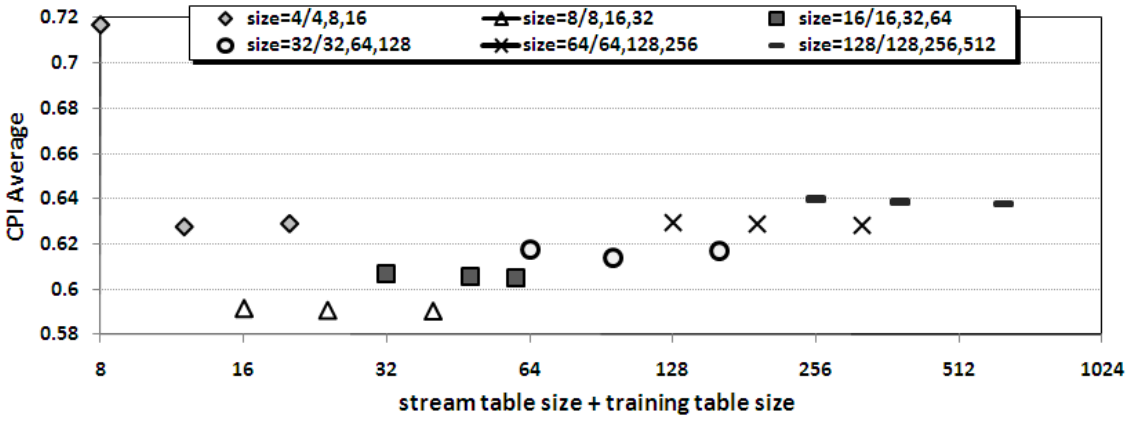

Figure 2-8. Sensitivity on stream history table sizes (Notation: size = stream table size / three training table sizes)

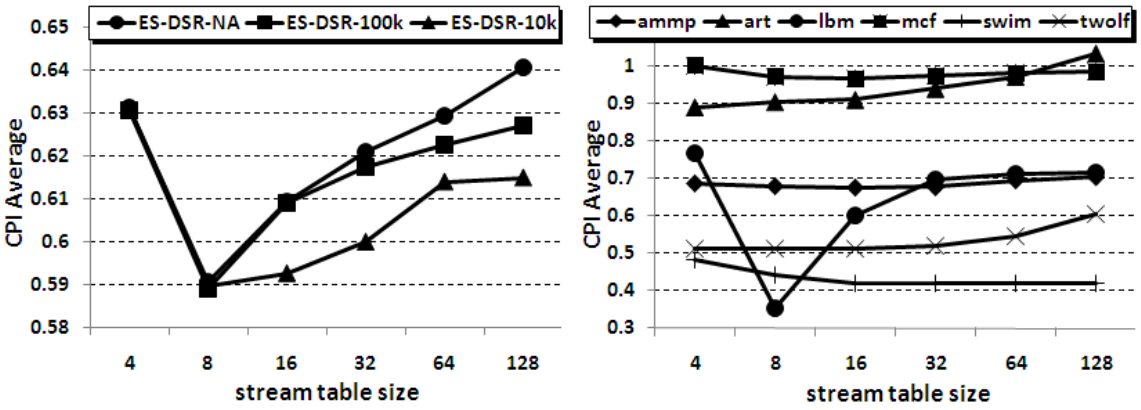

Figure 2-9. Overall (left figure) and individual (right figure) performance for Dead Stream Removal. ES-DSR-NA is without Dead Stream Removal, ES-DSR-100k/10k are with Dead Stream Removal and dead streams are defined as having been inactive for more than 100k/10k CPU cycles.

CHAPTER 3
SEMANTICS-AWARE, TIMELY PREFETCHING OF LINKED DATA STRUCTURE

3.1 Introduction

Applications often manifest a mixture of regular and irregular missing address patterns [36, 19]. One common occurrence of irregular missing addresses is when a program traverses through a tree-like Linked Data Structure (LDS), where one of the links in each node is selected for traversal to the next nodes [35, 36, 49, 15, 19]. Traversal through an LDS involves huge address spaces and irregular missing addresses, hence both regularity and correlation based prefetchers are not well suited for handling the LDS misses. However, given the fact that the missing address for traversal to the next node is always located in the current visiting node, LDS prefetches can be issued whenever the next link address is available. Nevertheless, due to a limited amount of work between visiting adjacent nodes, the LDS traversal creates a tight load-load dependent chain, often referred to as a pointer-chasing problem [36, 19]. Prefetching of the next node is delayed when accessing the current node encounters a cache miss.

Previous works proposed various hardware techniques to improve LDS prefetching. The dependence-based LDS prefetcher (DBP) [49] dynamically generates a compact representation for fast LDS traversal. It records all PC-based load-load dependences. Once a missing block comes back from memory, all dependent loads are triggered to shorten the delays. However, it suffers the difficult lateness in timely prefetching the needed LDS data. Figure 3-1 shows the IPC comparison of DBP and an ideal DBP in which the prefetch takes zero cycle to load the block. The results of twelve pointer-intensive applications show that on the average additional 52% of the IPC

improvement can be obtained if all the prefetching blocks can be ready without memory delays. This leaves a large room for improvement of the LDS prefetching.

In this chapter, we describe a semantics-aware hardware techniques to improve the lateness issue in LDS prefetching. This work makes two important contributions. First, to the best of our knowledge, this is the first attempt to dynamically establish the semantic structure of the traversal node by detecting the link traversal history. Three experimental rules are established to identify the pointer links for advancing the traversal to the next node for accurate LDS prefetching. Second, based on the semantic structure of the traversal node, we integrate three leap prefetching techniques for timely LDS prefetches. It incurs small storage overhead, minimum memory bandwidth requirement, little cache pollution, more timely prefetches, and most importantly it does not need program changes, compiler helps, and/or new ISA.

The remainder of this chapter is organized as follows. Section 3.2 describes the general characteristics of the LDS traversal. Section 3.3 presents mechanisms to detect the traversal links, and shows how to record the program semantics and use it to trigger the LDS prefetch. Section 3.4 presents the evaluation results. Related work is given in Section 3.5 followed by a brief summary in Section 3.6.

3.2 Characteristics of LDS Traversal

LDS traversal encounters a tight load-load dependence chain and prevents prefetchers from timely prefetching of the next traversal node. We examine several pointer-intensive workloads to understand the characteristics of LDS traversal and seek opportunities for breaking the tight load-load dependence chain. In this section, we present three hardware-based leap prefetching techniques illustrated with real examples. The first multi-link leap prefetching technique prefetches the adjacent nodes

through all pointer links in the current visiting node. The second multi-node leap prefetching technique prefetches the adjacent nodes through the pointer links of another node located in the same physical block of the current visiting node. The third pointer-stream leap prefetching technique prefetches the linked lists of nodes from a streaming pointer array. To examine the characteristics of the LDS traversal, we select function Refresh_Potential in mcf from SPEC2000 [57] (Figure 3-2) and HashLookup in mst from Olden [48] (Figure 3-5) for detailed analysis.

3.2.1 Leap Prefetch through Multiple Links:

refresh_potential has a general but sophisticated LDS traversal. Based on the depth-first traversal, Refresh_Potential traverses through the entire tree structure using one of the three pointer links, child, pred, and sibling. During visiting each node, the data link, basic_arc is used to access the data record. Each node structure has 60 bytes and the total tree size is about 4MB. Simulations show that given a 1MB cache, cache misses are frequently encountered in advancing to the next node using one of the pointer links as well as in accessing the data record through the basic_arc.

As seen from refresh_potential and other previous works [49, 50], the amount of work between LDS misses is limited. With the aggressive out-of-order execution, there is a bounded improvement from prefetching the next node upon returning the missing block of the current node. In the depth-first traversal, however, the traversal path backtracks through previous visited nodes once the depth search using the link child reaches to an end. This backtracking provides the first opportunity for leap prefetches as illustrated in Figure 3-3, where the traversal path goes through the sequence of nodes: A, B, C, D, C, B, E, F, G, F, and H using three pointer links: p1, p2 and p3.

For simplicity, we assume each node occupies a cache block and the first visit of node B encounters a miss. When block B is available, the LDS prefetches of nodes C and E are triggered through links p1 and p3 respectively. However, the subsequent traversal is to node C, hence node E is prefetched incorrectly. Nevertheless, after reaching to the end (node D) of the depth path, the traversal backtracks to visit node B again. In this second visit, the next traversal node becomes E. Therefore, the LDS prefetch through multiple links provide the opportunity to leap prefetch node E when node B is first visited. We refer this type of leap prefetch as a multi-link prefetch. Note that for clarity, we only draw the relevant traversal links. Due to limited work in each node, the prefetches of nodes C and D are late for the demand misses. The leap prefetch of E, on the other hand, can further prefetch F and G ahead of the traversal for alleviating the lateness issue.

3.2.2 Leap Prefetch through Multiple Nodes:

In LDS applications, the size of a logical node is often mismatched with the physical cache block. This mismatch provides the second opportunity for leap prefetches as illustrated in Figure 3-4. In this example, the traversal path goes through four nodes: A, B, C, and D using two pointer links: p1 and p2. The boundary of a logical node is marked by the thick dashed line in each physical block. Let's assume accessing node A is a miss. When the missing block is available, prefetches through p1 and p2 of the current node can be triggered as described before. In addition, the leap prefetches can also be initiated through the links of another node D located partially in the same missing block. In the example, the prefetch through D's links is useful when A is visited since D will be visited after traversal through nodes B and C. We refer this type of leap prefetches as a multi-node prefetch.

3.2.3 Leap Prefetch through Pointer Streams:

HashLookup in mst represents a different class of LDS traversal (Figure 3-5). HashLookup searches a hash table containing an array of pointers, each links to a list of nodes. Each node consists of a key, a pointer to the data record, and a pointer link to the next node. This function is invoked repeatedly to locate the node with a matching key. The key is first hashed to the pointer array before traversal to the selected linked list. Given that the short linked lists, leap prefetching subsequent nodes in each individual list has limited benefit.

From detailed analysis, we discovered that traversal through the pointer array exhibits a regular stream or stride behavior. Similarly, we observe another pointer array example in omnetpp06, the pointer array is scanned linearly and each pointer links only to a data record. With an integrated stream/stride prefetcher, the leap prefetching of future traversal lists can be initiated along with prefetching of the pointer array, referred as a pointer-stream prefetch. In an example in Figure 3-6, the traversal path is p1, A, B, C, p2, D, E, F, p3, and G. Assuming that accessing the pointer array has been identified as a stream and p2 is prefetched when p1 is accessed, prefetching to the next linked list of D, E, and F can start while the traversal is still on the p1 list.

3.3 Capture Node Semantics in LDS Traversal

The key requirement for capturing the LDS traversal is to dynamically identify the load (link) which updates the base register for advancing the traversal. Given the semantic structure of all the links in a node, accurate and timely prefetches to the subsequent nodes can be issued. Detailed description of the semantics-based LDS prefetcher will be given in Section 3.3.3. In this section, we describe the general rules and show examples in detecting the traversal links. There are two types of links in a

node, the link for advancing to the next node and the link for accessing the data record in the current visiting node, referred as a p-link and a d-link load respectively, also referred as recurrent and traversal loads in [49].

3.3.1 General Rule of Traversal Link

Based on several LDS-intensive applications, we define three experimental rules in identifying the traversal links. We define a load as a source load if it produces the base address for a dependent load. By definition, a source load is either a p-link or a d-link load. A non-source load is referred as a data load. A source load is a p-link load according to three rules.

- **Identity rule**: A source load is a p-link load if it has been identified as a p-link load previously.
- **Same-link rule**: For any two consecutive source loads with the same PC (i.e. two consecutive execution instances of the same source-load instruction), the load is a p-link load if the later load is a dependent of the early load.
- **Multiple-link rule**: A source load, whose parent source with a different PC is a previous identified p-link load, is also a p-link load if a dependent source load with the same or different PC as the parent source is also an identified p-link load.

The above experimental rules are based on the fact that when the traversal goes through one of the pointer links, the subsequent traversal link must follow a load-load dependence chain from the previous traversal link. The identity rule is obvious since the same link can be used repeatedly for advancing to the next node. The same-link rule and the multiple-link rule are applied when two consecutive node traversal through either the same or different links. In a traversal through the same link, the later source load must be a dependent of the early source load. However, when two consecutive source loads are from different links, the later source may not be a p-link load even if its base register is a dependent of the early p-link load. In this case, it is necessary to

check the subsequent dependent source load as described in the multiple-link rule. Since the same link can be used by different instructions, identifying a p-link can potentially be delayed.

3.3.2 Record Pointer Links

In this section, we describe the technique to detect and identify the p-link loads and d-link loads. A Memory Reference Buffer (MRB) is established to record the recent memory reference history. Besides the PC and the destination register contents, MRB also records offset of each instruction, type of a load (p-link/d-link/data/unknown), and a parent link to the last instruction which generates its base address as shown in Figure 3-7. All memory instructions excluding the stack instructions enter the MRB in order when they are committed. The parent link is established when an instruction enters MRB and finds a load-load dependence with a previous instruction.

mcf is used as an example to illustrate how to apply the experimental rules to identify p-link loads. From the code in Figure 3-2, there is a direct load-load dependence of child→child, where the parent and dependent instructions share the same PC. Such dependence can be detected in MRB as illustrated in Figure 3-7. According to Rule 2, the load of child (offset 12) is identified as p-link load. Similarly, pred→pred can also be detected and the load of pred (offset 8) is also a p-link load (not shown in Figure 3-7). Although direct sibling→sibling does not exist, there is an indirect pred→sibling→child. According to Rule 3, sibling (offset 16) is identified as a p-link load. These identified p-link loads will be updated in the MRB and the Node-ST which will be described next.

3.3.3 Node Semantic Structure and LDS Prefetch

Given the algorithm in capturing the traversal links, this section narrates how the traversal node structure is established, how the LDS prefetching is triggered, and how

the lateness in LDS prefetching can be improved. Besides the MRB, two additional structures for guiding the LDS prefetches are established. The Node Structure Table (Node-ST) records the detected semantic structure of traversal nodes and the Link PC History (PC-HT) table saves the PCs of the p-links for detecting any new LDS traversals as illustrated in Figure 3-8. Each entry in the Node-ST records the node structure for a particular tree traversal. Due to the unknown physical size of each node, we record a node with 64 bytes which matches the physical block size. Within each node, the data type of every 4 bytes is saved resulting in 16 recorded types. There are four data types for each 4 bytes including p-link, d-link, data, and unknown. The first three types are detected during the LDS traversal as described in the previous section. The unknown type is assigned to the corresponding 4 bytes which were untouched during the traversal. From the example trace in Section 3.3.2, the detected node structure is shown in the bottom of Figure 3-7 where the three p-links, prev, child, and sibling are recorded with offsets 8, 12, and 16; the d-link basic_arc has offset 32; and the data orientation and potential have offsets 28 and 44, respectively.

The PC-HT records the PCs for the detected p-link loads and the locations of the corresponding node structure in the Node-ST. Note that since the number of different LDS traversal is small, only a few entries are required in both Node-ST and PC-HT. In addition, the PC-HT can be organized as a set-associative table to further reduce the searches. The corresponding PCs need to be invalidated from the PC-HT when a node structure is replaced from the Node-ST.

Furthermore, additional LDS information including the base address of the next traversal block, the location of the node structure in the Node-ST, and the level of the

LDS prefetches are recorded in the MSHR (or the prefetch queue) for the LDS misses sent to the memory. Once the missing block comes back, the recorded information can be used to trigger further LDS prefetches with minimum delays. The detailed procedure is now described.

Capturing and recording node structures: A new LDS traversal begins when a detected p-link load neither has a source load in the MRB nor has its PC recorded in the PC-HT. In this case, a new entry to record the node structure is created in the Node-ST to replace an old node structure in the LRU position. Meanwhile, the PC of the p-link is inserted into the set-associative PC-HT. Any subsequent p-link, d-link, and data loads detected through the MRB will be used to update the corresponding types of the node structure located in the MRU position in the Node-ST. The PCs of all detected p-link are inserted into the PC-HT for determining any new LDS traversal.

Initiate LDS prefetches: When a p-link load is detected from the MRB, the destination address of the load is the base of the subsequent node traversing from the detected p-link. The corresponding node structure can be obtained from the Node-ST. From the offsets of the node structure, the physical block(s) of the subsequent node can be calculated based on the largest valid offset value. Each node can be located in more than one block due to un-alignment between the physical block and logical node. All the blocks for the subsequent node is then prefetched if the blocks are not located in cache and do not belong to an active missing block recorded in the MSHR. Note that the latter case occurs frequently since accessing the p-link for traversal to the next node is usually the last instruction in the visiting node. In Figure 3-7, for instance, node→orientation causes the first miss in the visiting node. The base of the subsequent

node and the location of the node structure are attached in the MSHR for triggering further LDS prefetches when the missing block comes from memory.

Chain of LDS prefetches: When a missing block comes from memory, the information attached in the MSHR or the prefetch queue indicates if the missing block is a LDS miss and whether the LDS prefetch should continue based on the prefetch level. To continue the LDS prefetches, the base address of the subsequent node and the structure of the node can be used to locate the pointer links for the node in the returned missing block. We can then calculate the base address for the follow-on node through each p-link as well as the number of blocks for the follow-on node. These blocks can then be prefetched in the same way as in the initial LDS prefetch. When issuing the prefetches through the p-links, the attached prefetch level in the MSHR/prefetch queue is incremented for the depth of the prefetch chain. The LDS prefetches stops when certain threshold level is reached. Note that a single level of prefetch through the d-link can also be triggered for prefetching the data block.

Leap LDS prefetches: Recall that leap prefetches are enabled according to three special characteristics of the LDS traversal. In the first type of multi-link prefetch, the LDS prefetches are triggered from all the p-links recorded in the node structure as described in triggering a chain of LDS prefetches. Meanwhile, in searching for any companion node in the missing block, the content of the block is first scanned in determining the data type of each 4-byte. Similar to the technique used in the content-directed prefetcher [16], we match the high-order bits with the address of the block for pointer links. Given the structure of the node, and the scanned type for each 4-type in the physical block, the pointer links for any companion node in the returned missing

block can be located. In case a companion node is found, the base addresses of the follow-on nodes from the companion p-links can be calculated for issuing the multi-node prefetches.

In triggering the pointer-stream prefetches, the regular miss pattern in the pointer array is detected and prefetched first by an integrated stream/stride prefetcher. Meanwhile, the source pointer load for each linked list can be detected from the MRB as a p-link load. Once detected, the source load address is searched among the recorded streams. In case that the link address matches a trained stream, the offset of the link load is attached to the trained stream. When a prefetched streaming block arrives from memory, the recorded offset along with the stride are used to calculate the base addresses of all associated linked lists for triggering the pointer-stream prefetches.

3.4 Performance Results

We simulate and compare the performance of seven L2 prefetching methods including an enhanced stream/stride prefetcher (Stream) [58, 60], the dependence-based prefetcher (DBP) [49], the content-directed prefetcher (CDP) [16], a pointer-stream enhanced leap DBP prefetcher (DBP-leap), and three variations of the semantics-aware leap prefetchers. To compare the effectiveness of the leap prefetching techniques, we simulate the semantics-aware prefetchers with the multi-link prefetch only (Semantics-leap1), with both multi-link and multi-node prefetches (Semantics-leap2), and with all three leap prefetching schemes (Semantics-leap3). Note that by triggering all dependent loads, DBP accomplishes similar leap prefetches as that using the multi-link prefetch. DBP can also take advantages of pointer-stream prefetches to trigger multiple LDS streams. All the simulated LDS prefetchers integrate an enhanced stream/stride prefetcher to handle regular miss patterns.

The simulation methodology is described in Section 1.4, parameters for the simulator are listed in Table 1-2. For each scheme evaluated, specific parameters are given in Figure 3-9. We simulate DBP, CDP and Semantics with the same prefetch queue size and prefetch depth. Other parameters are chosen closely to those used in the original papers. For stream, we simulate an aggressive configuration with the prefetch distance of 32 and degree of 4. For Semantics, we use a small MRB of 32 entries with a 4-entry Node-ST and a 32-entry, 4-way associative PC-HT.

3.4.1 IPC Comparison

Figure 3-9 shows the normalized IPC with respect to baseline processor without any prefetching (no-prefetch) for the seven prefetching methods. We can make several important observations. First, among the LDS prefetchers, Semantics-leap3 shows about 45% average performance improvement over no-prefetch, 20% over stream, 16% over DBP, 5% over DBP-leap and 23% better than CDP. Semantics-leap3 has significant performance improvement over Stream, DBP and CDP in mcf06, omnetpp06, xalanc06, mcf00, health, and mst, while is relatively the same in gcc06, perl06, ammp, parser, vpr and bisort. Among all the applications, mcf06, perl06, xalanc06, mcf00, bisort and health respond reasonably well to the leap prefetching techniques, while gcc06, ammp, parser and vpr do not show much impacts on prefetching. Also noted is that pointer-stream leap prefetching benefits both DBP-leap and Semantics-leap3 significantly in omnetpp06 and mst.

Second, in comparison with the leap prefetching techniques, Semantics-leap1 with only multi-link leap prefetching improves about 26% IPCs over no-prefetch. With additional multi-node leap prefetching, the improvement of Semantics-leap2 goes up to 32%. With all three techniques, the Semantics-leap3 shows an impressive 45%

improvement. The improvement from the pointer-stream leap prefetching stands out in omnetpp06 and mst with 1.8 and 2.2 times of IPCs. Both applications have a regular streaming based access pattern to a large pointer array with each entry linking to a short list of nodes or a data record. The multi-node leap prefetching, on the other hand, has a big improvement in mcf06, mcf00 and health, but is rather insensitive in others.

Third, DBP generally performs well as long as the semantics-aware prefetcher performs well and is very close to Semantics-leap1. DBP triggers prefetches based on all load-load dependences which is effectively the same as Semantics-leap1. However, Semantics-leap1 has a slight edge in timely prefetching by using the semantic structure of node since certain links or data in a node may not encounter direct load-load dependence from all other p-links.

Fourth, among all the method, CDP is extremely effective in health, but performs poorly in most of other applications. Health traverses through a double linked list with small node size. The list is created dynamically, hence the traversal does not form a sequential stream. However, adjacent nodes are located in a small region. Therefore, prefetching based on all pointers in a block is very accurate with small number of prefetched blocks. For other applications, greedy prefetching based on all pointers generates excessive and inaccurate prefetches. It shows negative impacts in gcc06, mcf06, omnetpp06, xalanc06, mcf00 and vpr.

3.4.2 MPKI Comparison

Figure 3-10 summarizes MPKI of the seven prefetch schemes normalized to no-prefetch. Overall, Semantics-leap3 has the lowest average MPKI, followed by DBP-leap,CDP, Semantics-leap2, Semantics-leap1, DBP and Stream, in which Semantics-leap1, and DBP have similar MPKI. These MPKI results are generally consistent with

the IPC improvement in Figure 3-9 with one exception: CDP. CDP has lower average MPKI mainly because of health. Its excessive memory traffic causes delays to other memory requests, hence has the worst IPC improvement. Note that the MPKI does not include the partial hits to the block already issued to the memory, which also affect the overall IPC performance. Discussions about the partial hits will be given in the next subsection.

3.4.3 Prefetch Accuracy, Coverage and Extra Traffic

In Figure 3-11, each performance bar represents the entire memory traffic broken down into four categories, miss, prefetch partial hit, prefetch hit, and useless prefetch. A memory request encounters a prefetch partial hit if the requested block is on the way back from the memory by an early prefetch. Although the partial-hit block is not in the cache, its penalty cycles are reduced. Useless prefetches are prefetched blocks that are replaced before any usage. The prefetch accuracy is measured by the ratio of the total prefetch hits and partial hits divided by the total number of prefetches. The miss coverage is the percentage of the miss reduction from the misses without any prefetch. Note that the miss coverage is a net result of the miss reduction from prefetches and the miss increase due to cache pollution.

We can make a few observations from this figure. The first and most obvious observation is the excessive memory traffic generated by the inaccurate prefetching of CDP. Except for health, greedy prefetching through all pointers in a block demands excessive memory bandwidth along with the cache pollution that can nullify any benefit of the prefetch hits and partial hits.

Second, in general, DBP and semantics-aware prefetchers do not create much extra traffic where DBP has less traffic except for gcc06, xalanc06 and ammp. Three

leap prefetching techniques turn misses into partial hits or hits to reduce the penalty cycles with minimum impacts on the cache pollution. The pointer-stream leap prefetching is especially effective in mst and omnetpp06 with 74% and 65% miss coverage and 91% and 97% accuracy respectively. For xalanc06, the pointer-stream leap prefetching also successfully cover 47% of the misses, but the accuracy is only 69% resulting in a total of 29% extra traffic.

Third, the results of accuracy, coverage and extra memory traffic is consistent with the observation that mcf06, perl06, xalanc06, mcf00, bisort and health respond well to the prefetching techniques, while gcc06, ammp and vpr do not show much improvement by prefetching. Those applications that benefit from prefetching show high miss coverage. The applications without much benefit from prefetching have either low miss coverage in vpr or inaccurate prefetching in gcc06, xalanc06 and ammp.

3.4.4 Sensitivity Studies

In this section, we present the sensitivity studies of MRB size, Node-ST/PC-HT size, prefetch queue size, prefetch depth, memory latency, and distance/degree of stream prefetching. The default values of these six parameters are 32, 4/32, 64, 4, 200, and 32/4 respectively. While simulating different values for the sensitivity of a particular parameter, we set other parameters to the default value. For clarity, we only plot the geometric means of the respective prefetching methods.

MRB and Node-ST/PC-HT Size. As expected in Figure 3-12, the required sizes for the MRB, the Node-ST, and the PC-HT for accurately detecting the link traversal history are very small. A MRB with 16 entries can capture the link traversal history accurately. Larger MRBs make very little improvement. In simulating the Node-ST and PC-HT sizes, we always use a PC-HT that is 8 times size of the Node-ST. From the

results, a 2-entry Node-ST is sufficient. This is due to the fact that even if the number of different LDS traversals is more than the Node-ST entries, re-establishing a node semantic structure only takes a few instructions, hence the penalty is small. In our simulation, we set the default size of 32 for the MRB and 4 for the Node-ST.

Prefetch Queue Size and Prefetch Depth . Aggressive leap prefetches benefit from large prefetch queues as shown in the left of Figure 3-13. However, the benefit starts leveling off with a 32-entry queue. In our simulation, we select a 64-entry prefetch queue in order to minimize dropping of prefetch requests. It is interesting to see that given a prefetch queue larger than 16, the IPC actually decreases for CDP. This is due to the fact that dropping excessive prefetches in CDP can actually help the performance. For the depth of prefetch, deepening the prefetch level generally improve the performance. We choose 4 levels of prefetch in our simulation.

Stream Aggressiveness and Memory Latency . Moderate and aggressive stream prefetching with distance and degree of (16,2) and (32,4) have the highest IPC improvement as shown on the left of Figure 3-14. In Semantics-leap3, stream prefetcher not only helps prefetch blocks with regular missing address patterns, but also impacts the pointer-stream leap prefetching. Now consider the memory latencies, the higher the latency, the more performance improvement can be obtained from DBP and Semantics-leap3 as shown on the right side of Figure 3-14. With 500-cycle memory latency, the average performance improvement for Semantics-leap3 increases to 8% over DBP-leap, and 98% over CDP. In our simulation, we simulate an aggressive stream prefetcher of (32,4) with memory latency of 200 cycles.

3.5 Related Work

There are three works most related to the proposed semantics-aware prefetcher including the dependence-based prefetcher (DBP) [49], the content-directed prefetcher (CDP) [16], and the compiler-guided hybrid prefetcher (CHP) [19]. DBP dynamically generates compact representation for specialized pointer-traversal hardware. It records all PC-based load-load dependences to trigger the dependent loads upon available of the missing block from the source PC. The semantics-aware prefetcher with multi-link leap prefetching is very similar to DBP. However, in a few applications, a certain data record in a node is not a direct load-load dependent of certain pointer links, hence cannot be triggered directly by DBP. Both semantics-aware and DBP prefetchers respond well with the pointer-stream leap prefetching. But, semantics-aware has an edge over the DBP using the multi-node leap prefetching.

CDP [16] uses a historyless approach for LDS prefetching. Instead of recording the miss addresses, it searches for all the pointers from the content of a returned missing block to continue the prefetching. Semantics-aware prefetcher borrows the technique from CDP to identify the pointers in a physical block. Nevertheless, semantics-aware prefetcher dynamically traces the link traversal history to establish the semantic structure of the node for accurate LDS prefetching to overcome the main weakness of CDP. In addition, the leap prefetching based on a streaming pointer array is beyond the scope of CDP.

CHP [19] overcomes the inaccuracy weakness of CDP through compiler generated profiling information to guide the LDS prefetching. It requires inserting prefetch hints in the binary code based on the usage frequency of individual links. Along with an integrated stream prefetcher and certain throttling techniques in managing the

two prefetchers, they demonstrated significant performance improvement. We view the goals of CHP and semantics-aware prefetcher as similar given the fact that both want to improve the accuracy of CDP. However, instead of seeking compiler helps, semantics-aware prefetcher accomplishes the goal by dynamic tracing the link traversal history. Furthermore, the additional leap techniques in semantics-aware prefetcher are beyond what CHP can handle.

Besides DBP, CDP, and CHP, there are a few more works that have tried to improve LDS prefetching. The greedy prefetcher [41] launches the next node/nodes prefetching by inserting the prefetch instruction at the start of each loop iteration. The guided region prefetcher [64] uses compiler analysis to insert hints to prefetch through all link pointers in each node.

3.6 Summary

In this chapter, we present a semantics-aware prefetcher for accurate and timely Linked Data Structure prefetching. The semantics-aware prefetcher is a pure hardware solution. It dynamically establishes node structure of LDS through simple experimental rules based on the LDS traversal history. We also introduce three leap prefetch techniques to improve the lateness problem existing on earlier works. The evaluations based on several LDS-intensive applications have demonstrated the effectiveness of the proposed method.

Table 3-1. Simulation parameters

Prefetcher	Configuration
Stream	number of streams: 16, prefetch distance/degree: 32/4, 1-cycle latency
DBP	PPW window: 32, CT correlation: 256, depth: 4, 5-cycle latency
CDP	compare/filter/align bits: 8/4/1, search step: 2 bytes, depth: 4, 10-cycle latency
Semantics	MRB: 32, Node-ST: 4, PC-HT: 32, 4-way, depth: 4, 5 cycles for multi-link prefetch, 10 cycles for multi-node prefetch;

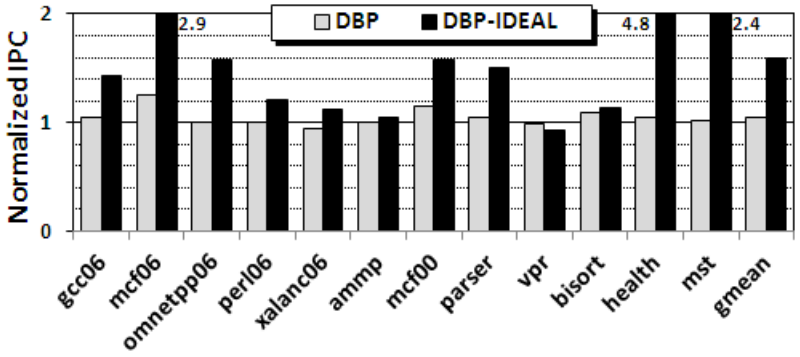

Figure 3-1. IPC comparison of DBP

```
Long refresh_potential (network_t *net) {
      node = root->child;
      while (node != root) {
           while (node) {
                if (node->orientation == UP)
                     node->potential=node->basic_arc->cost
                                     +node->pred->potential;
                else {…}
                     tmp = node; node = node->child;
           }
           node = tmp;
           while (node->pred) {
                tmp = node->sibling;
                if (tmp) {node = tmp; break;}
                else node = node->pred;
           }
      }
      return checksum;
}
```

Figure 3-2. An example of tree traversal from mcf

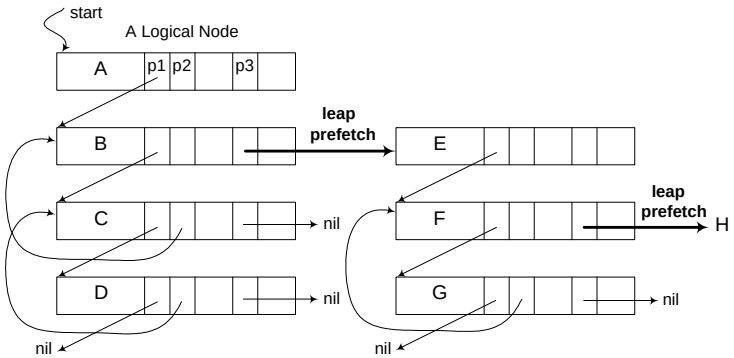

Figure 3-3. An abstract example for leap prefetching through multiple links in a node where the traversal path is A→B→C→D→C→B→E→F→G→F→H

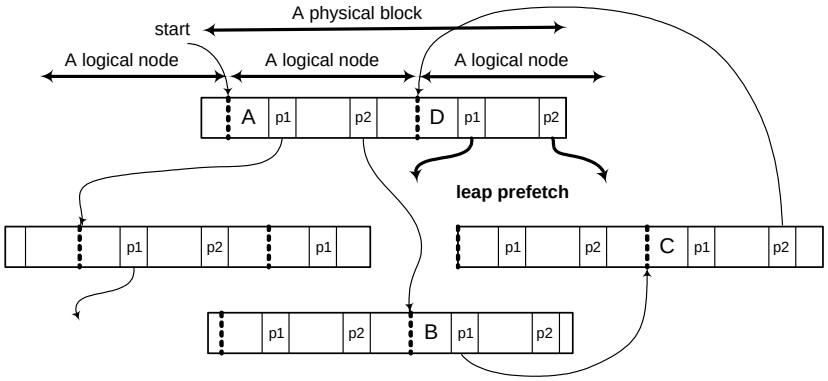

Figure 3-4. An abstract example for leap prefetching through multiple nodes in a block where the traversal path is A→B→C→D

```
void *HashLookup(int key, Hash hash) {
      j = (hash->mapfunc)(key);
        ...
      for(ent = hash->array[j];ent->key!=key; ent=ent->next);
           if (ent)
                return ent->entry;
}
```

Figure 3-5. An example of LDS traversal from mst

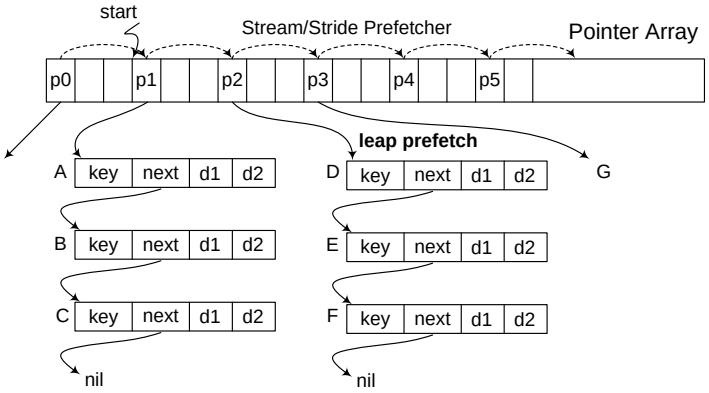

Figure 3-6. An abstract example for leap prefetching through a pointer stream where the traversal path is p1→A→B→C--->p2→D→E→F--->p3→G

PC	Inst	[Dest]	Parent	Offset	Type
b6c6e4e0	**mov eax,12[ecx]**	b7d875c8		**12(child)**	**p-link**
b6c6e4cb	cmp 28[ecx]	---		28(orient)	data
b6c6e4d1	mov eax,8[ecx]	b7e79d0c		8(pred)	d-link
b6c6e4d4	mov edx,32[ecx]	b2094518		32(basic_arc)	d-link
b6c6e4d7	mov eax,44[eax]	2e		44(potential)	data
b6c6e4da	add eax,16[edx]	feced32e		16(cost)	unknown
b6c6e4dd	mov 44[ecx],eax	---		44(potential)	data
b6c6e4e0	**mov eax,12[ecx]**	b7e6bf34		**12(child)**	**p-link**

Figure 3-7. The Memory Reference Buffer (MRB) for tracing inner “while (node)” loop in refresh_potential

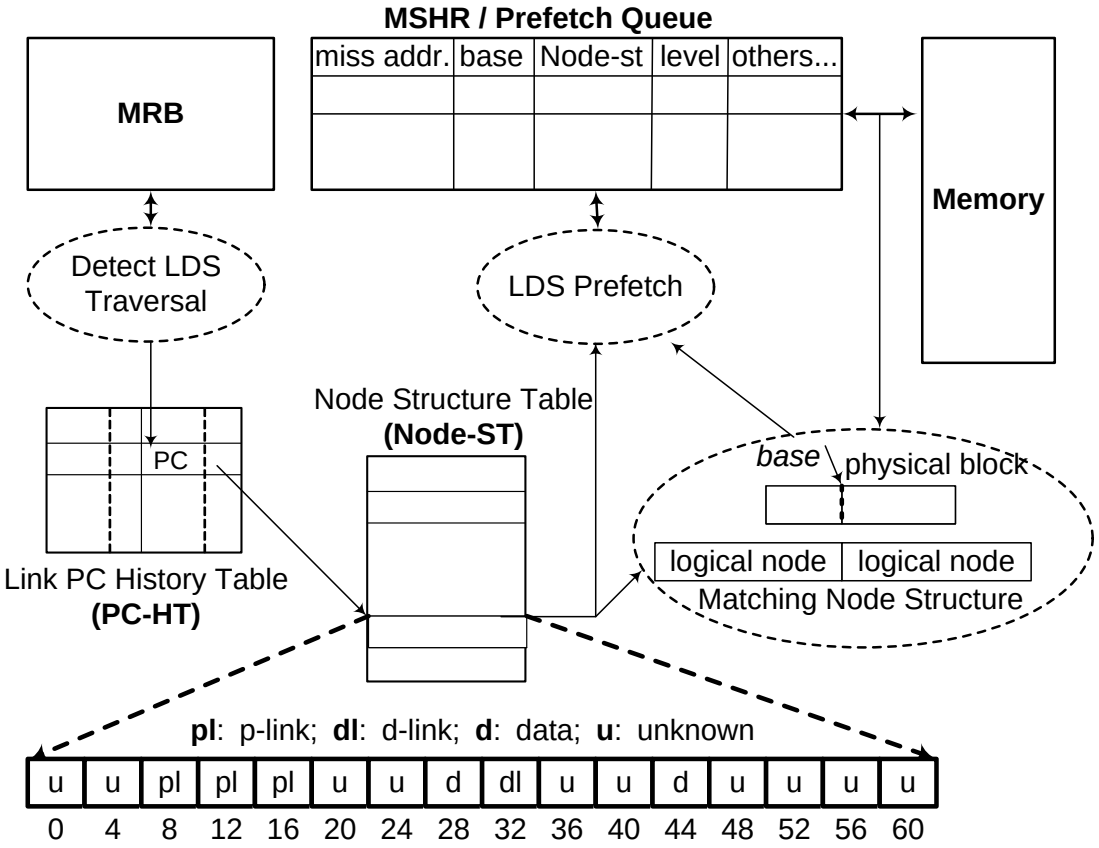

Figure 3-8. The basic structure of LDS prefetching

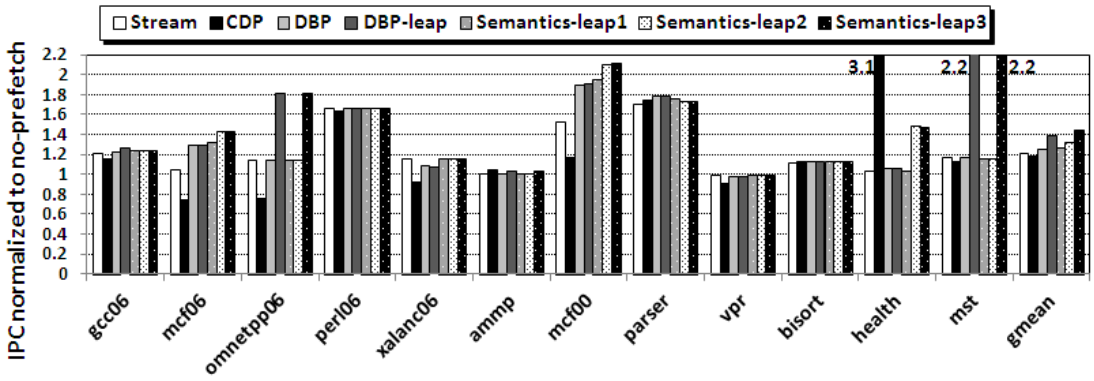

Figure 3-9. IPC comparison of seven prefetching methods

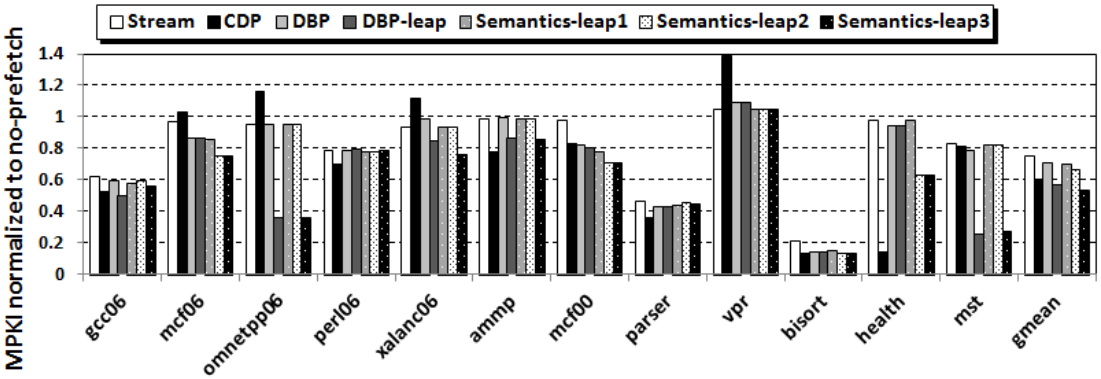

Figure 3-10. Misses-Per-Kilo-Instruction (MPKI) for the seven prefetching methods

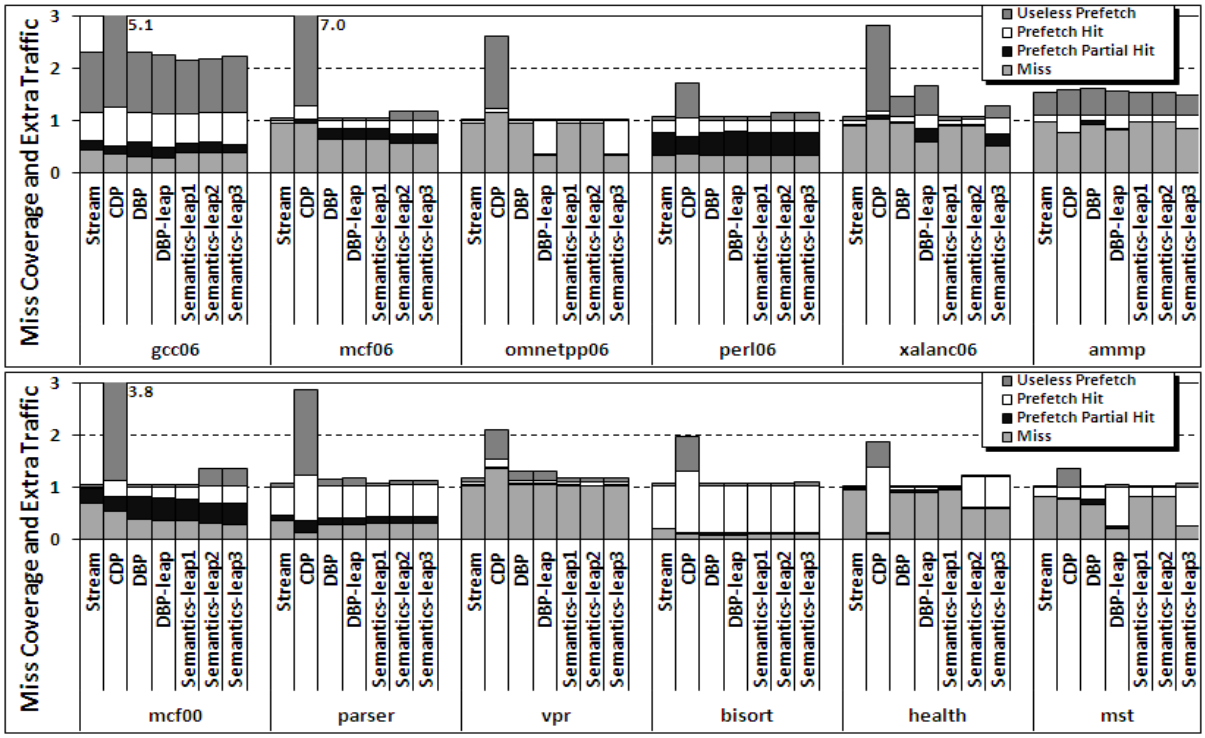

Figure 3-11. Miss coverage and extra traffic

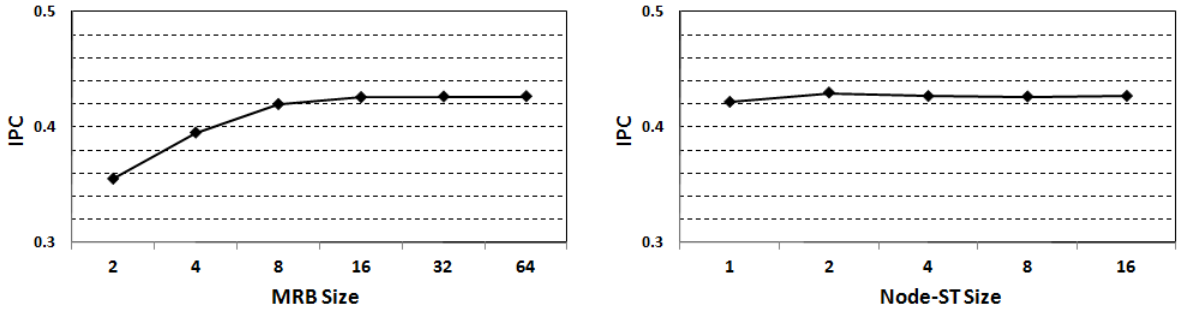

Figure 3-12. Sensitivity studies on MRB and Node-ST/PC-HT sizes for Semantics-leap3

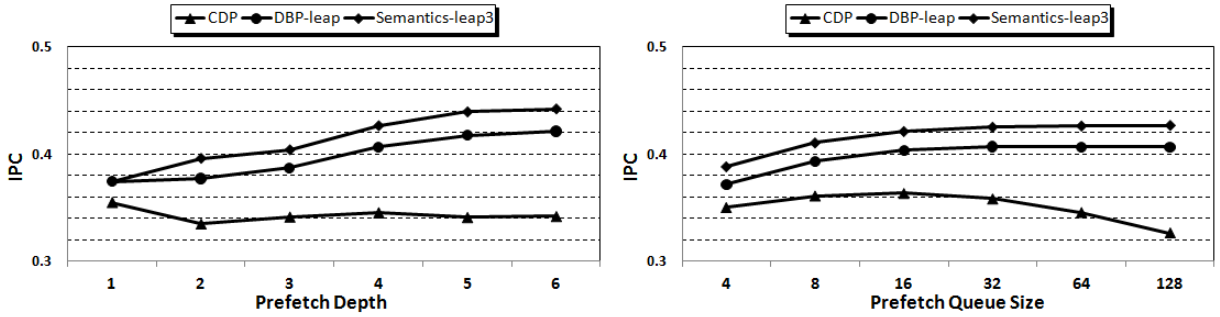

Figure 3-13. Sensitivity studies on prefetch queue size and depth

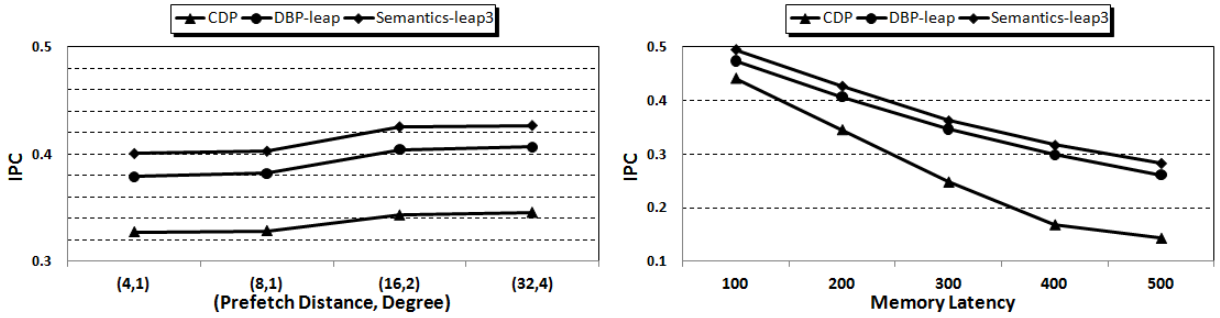

Figure 3-14. Sensitivity studies on aggressiveness of stream prefetcher and memory latency

CHAPTER 4
ENCODING PER-BLOCK MISS CORRELATIONS IN COMPRESSED DRAM FOR DATA PREFETCHING

4.1 Introduction

Cache misses frequently exhibit repeated streaming behavior, i.e. a sequence of cache misses has a high tendency of being repeated [13, 65, 55, 14, 66, 56, 67]. To prefetch these correlated misses, correlation-based prefetchers record the nearby missing address history to trigger prefetches assuming such miss correlations will be repeated. To be effective, however, correlated prefetchers must record long cache miss history and incur significant storage overhead. Recent solutions store the streams of miss addresses in off-chip DRAMs [54, 65, 14, 22, 56, 67]. For example, the Sampled Temporal Memory Straming (STMS) prefetcher [56, 67] uses an off-chip circular buffer to record the sequence of recent miss addresses. In addition, an off-chip hash-based index table records the head addresses of the repeated streams with a pointer to the stream location in the circular buffer. Collectively, we refer the information recorded in two tables as the miss-stream metadata. When the head of a missing stream reappears, the repeated stream can be detected and the missing addresses in the circular buffer are fetched to trigger prefetches. Although STMS achieves high accuracy and miss coverage in capturing repeated missing streams, the latency and memory bandwidth requirement for extra off-chip accesses of the metadata can degrade the benefit of data prefetching [67]. This degradation will become more serious as the number of cores per die continue to increases while the external memory bandwidth is constrained by the number of package pins [28].

In this chapter, we investigate a new miss-correlation prefetcher which can eliminate the extra off-chip access. Similar to the original pair-wise correlated prefetcher [9], the new miss-correlation prefetcher establishes the miss correlation for every memory block, hence is referred as a per-block miss correlation prefetcher. Different from the original pair-wise correlated prefetcher, the proposed prefetcher is based on simple space and time proximity and we called it space-time miss-correlation prefetcher. Each miss can have a number of correlated misses which are close to the parent miss both in time and space. A small FIFO history buffer is used to record the recent misses [43]. All misses in the history buffer are time correlated to each early miss in the buffer. To qualify for the space correlation, the addresses of the correlated misses must be within a certain range from the parent miss.

Recording per-block miss correlations can provide unbounded correlation history. However, it incurs significant space overhead and extra latency/bandwidth in accessing the miss-correlation metadata. In this chapter, we investigate a novel approach to encode the correlation in DRAM with the data of the block. Such miss correlations can be fetched automatically along with the content of a miss block. It requires zero additional access to obtain the miss correlation for triggering prefetches, hence we refer this novel approach as miss-correlation folding.

The solution for the space explosion problem in recording the per-block miss correlation become feasible due to recent advances in data compression techniques [68, 76, 61, 71, 20, 72, 2, 3]. Compression techniques store compressed data in caches and DRAMs, thereby increase the effective cache and memory sizes for holding more blocks. In addition, by transferring compressed blocks between caches and memories,

the effective memory bandwidth can be improved [33]. In contrast to the existing memory compressions for packing more blocks in DRAM with variable block sizes, our approach is to simply compress individual memory blocks to free up space for encoding per-block miss correlation while maintaining the original block size. We examine several compression algorithms [68, 76, 20, 72, 2] and pick one which can create sufficient free space in each memory block. A few researchers studied the relationship between data compression and prefetches [63, 17, 77, 2, 3]. However, to the best of our knowledge, the per-block miss correlation prefetcher is the first attempt to use DRAM compression techniques to create free space for encoding the prefetch history.

The remainder of this chapter is organized as follows. We first collect the MPKI distributions from the selected workloads and demonstrate a wide-range of MPKI exists during each program execution in Section 4.2. We also show the performance gap between two prefetchers when the prefetching metadata is located either on-chip or off-chip. In Section 4.3, we show the potential miss coverage using the per-block miss correlation for prefetching. In Section 4.4, we describe an existing compression algorithm and show its effectiveness in creating free bits in each memory block. In Section 4.5, the detailed algorithm for establishing per-block miss correlations with limited free bits and the basic architecture design of encoding per-block miss correlations in DRAM are given. This is followed by the performance evaluation methodology and results in Section 4.6. Related works are given in Section 4.7 and the conclusion is in Section 4.8.

4.2 Prefetching under High MPKI

Applications display repeated missing streams which can be captured for accurate data prefetching. However, as illustrated in Figure 4-1, for the prefetches to be effective,

the prefetch stream must run sufficient ahead of the execution stream. Adequate memory bandwidth is required to deliver the data to the execution stream as well as the prefetch stream. The amount of memory bandwidth required can be determined by the IPC and the MPKI during program execution. In future many-core CMPs, concurrent execution threads are expected to deliver higher overall IPCs. Meanwhile, applications often exhibit a wide range of MPKI – Figure 4-2 shows accumulated MPKI ranged from 0 to 120 for a set of parallel workloads. During program execution, the regions where high IPC and high MPKI coincide, memory bandwidth can be saturated. In additional to the prefetch stream, if the prefetcher requires accessing and updating metadata off-chip, supporting the metadata traffic at these high bandwidth regions can be challenging as it competes for limited memory bandwidth.

To understand the performance impact on placing the miss-stream metadata off-chip, we compare the IPC improvement over no prefetch of the two STMS prefetchers – one places the metadata off-chip and the other fits the metadata on-chip. Placing the metadata on-chip requires zero cycle delay and no off-chip traffic in detecting and accessing the miss-stream metadata. To make STMS prefetchers more effective, we double the size of the circular buffer and the index table reported in [67]. We also integrate a regular stream/stride prefetcher to handle regular missing addresses. Figure 4-3 shows a significant IPC gap exists between the two prefetchers. swim, ocean, and facesim are the exceptions as they contain high MPKI regions but have little or no IPC difference for the two STMS prefetchers. This is because these workloads are dominated by regular streams; hence, there is little metadata traffic off-chip. None-the-less, this data confirms our belief that supporting off-chip prefetch metadata will be

challenging as external memory bandwidth is more stressed for multithreaded concurrent execution.

4.3 Miss Coverage Using Per-Block Miss Correlation

To motivate our work, we evaluate the miss coverage of a per-block space-time prefetcher with an unbounded correlation history. We consider that a miss is correlated with an early miss when the two misses are closely encountered both in time and space. To capture the time correlation, we implement a FIFO-based Miss History Buffer (MHB) to record the recent cache misses. When a cache miss occurs, it searches all previous misses in the MHB for identifying the misses to whom the new miss is correlated. A new miss is a correlated miss if the distance from the parent miss can be encoded by a fixed number of bits. All found correlated parents in the MHB search for their correlated misses recorded in DRAM. A new miss is covered if it has been recorded as a correlated miss in one or more parent's DRAM block. In this case, the new miss can be prefetched by the miss of the correlated parent. The miss is uncovered if searching for the correlation fails. In this case, a new correlated miss is encoded in all parents' DRAM blocks.

The size of the MHB and the number of bits to encode the distance determine the aggressiveness in capturing the miss correlation. In the experiment of potential miss coverage, we simulate a MHB with 128 entries and use 15 bits to encode the correlated block distance. We assume each DRAM block can record up to 16 correlated misses. It is important to note that in collecting the potential miss coverage, the actual prefetch does not take places, hence no cache pollution and prefetch timeliness is considered. Therefore, the result shows an optimistic upper bound in the miss coverage. Figure 4-4

shows the per-block miss-correlation can cover 57% to 100% of all misses with most workloads greater than 80% coverage.

4.4 Compression Schemes

In this work, we consider a novel usage of the data compression techniques to provide space for encoding per-block miss correlation history. Existing memory compression methods such as IBM MXT technology [1, 61] compress the entire main memory with variable block size to achieve the highest compression ratio. However, study shows that the overall IPC improvement with compressed memory for running SPEC2000 benchmarks is merely 1.3% [1].

Unlike IBM MXT, we look for fast and efficient compression algorithms which work well with small memory blocks (e.g. 64 bytes). We consider a memory block compressible if the block has f or more free bits after compression, where f is the number of pre-determined free bits for encoding the prefetching information. The compression coverage is the ratio of compressible blocks in the entire memory. It is important to note that 100% compression coverage is unnecessary for prefetching. The effect is that missing prefetch metadata for non-compressible blocks degrades prefetch performance.

Our implementation chooses the Single-Value Dynamic Encoding (SVDE) compression algorithm [47] (Figure 4-5). The SVDE compresses each block independently. It can work with 16-bit or 32-bit words. SVDE scans for a word that has k matches within the block. In the case there are multiple words that meet this criterion, only the first word is selected. The selected word is stored in the dictionary field in the compressed block. To increase the probability of matches, an m-bit mask is used to mask off the lowest m-bit in the words during comparisons. For each word that matches

the dictionary word, an n-bit index (where n is log2 of the number of words in the memory block) and an m-bit masked value is stored.

The number of bits saved by SVDE can be computed as:

$$(k-1)*(w-m)-k*n-1$$

For configuration of w=16, k=4, m=4, n=5, the number of bits saved is 15.

Decompression works as follows. If the block is not compressed (indicated by a '0' in the compression bit), then the whole block is read. If a block is compressed (indicated by a '1' the compression bit), the dictionary word is read first. Then, each of the matched word is read and combined with the dictionary word to form the initial word. The recovered word is put in the appropriate location of the return buffer based on the stored index. For example, if the index is 3, then the recovered word is put in the 3rd position of the return buffer. Once all the matched words are recovered, SVDE then read each of the uncompressed words and put them into the return buffer in sequential order.

Table 4-1 shows the percentage of cache blocks that is compressible to provide 15-bit free space. Eight out of ten workloads have greater than 95% coverage while equake has 41% and facesim has 12% coverage. As mentioned before, low compression coverage could reduce the effectiveness of prefetch. Section 4-7 will discuss the prefetch performance of our scheme taking into account of compression coverage.

4.5 Establishing Per-Block Miss Correlation

Although per-block miss correlations can provide an unbounded history, it is constrained by limited free bits to encode the correlations. In this section, we describe

the detailed algorithm for establishing per-block miss correlations and the basic design of encoding per-block miss correlations in DRAM. Given f free bits after compression for each DRAM block, the per-block space-time miss correlation must be packed into the available f bits. We refer each DRAM block that records the miss correlation as the base (or parent) block. Two forms of encoding the miss correlation are considered. First, a straight-forward method is to encode the block distance from the base block to the correlated miss block. For timely prefetching, the distance is for the d-th miss followed the miss of the base block, where d is the prefetch distance. Given f available bits, the correlated miss must locate within the region of $\pm 2^{f-1}$ blocks from the base block. Second, to capture spatial correlation behavior, a bitmap vector can be used to encode up to f correlated miss blocks that are located within the same spatial region, where each bit represents a correlated neighbor. The spatial region is centered around the base block from $B - f/2$ to $B + f/2$, where B is the address of the base block.

The bitmap encoding method extends the per-block miss correlation from a single correlated miss up to f correlated misses without any specific order. In addition, unlike the previous approach [55] in which a fixed memory region must be aligned in the region boundary, per-block correlated regions are overlapped among neighbor regions to form a logically unbounded region. Such an unbounded region eliminates per-region triggering miss to initiate the spatial region prefetches. The distance correlation, on the other hand, remedies the small spatial region size by recording a single distance to extend the correlation to a wider region. The benefit of integrating the two miss correlations will be given in Section 4-6.

The basic architecture design of a per-block miss-correlation prefetcher is depicted in Figure 4-6. To capture per-block miss correlation, a small FIFO-based Miss History Buffer (MHB) records the recent cache misses. Each entry in the MHB saves the miss address and the detected miss correlations. Note that in capturing a repeated miss correlation, a prefetch hit is also placed in the MHB. The MHB can be multiple banked to provide sufficient bandwidth. When a block is aged out from the MHB, the miss correlation is saved in the cache temporarily with less than 2% space overhead and eventually moved to DRAM when the block is replaced. This miss correlation triggers prefetches when the block is missed again and fetched from DRAM. In throttling the prefetch, the first hit to a prefetched block initiates further prefetches using the encoded information in the prefetched block. A prefetch is dropped when the prefetch queue is full. The basic algorithm in detecting the miss correlation follows several steps.

- When a cache miss or a prefetch hit occurs, the block address enters the MHB. Searching for a bitmap or a distance-correlation parent is carried out throughout the MHB. Such searches are not timing critical and can be performed sequentially in a pipelined fashion.
- When the new block address falls into the bitmap region of any previous miss, the correlated bitmap of the previous neighbor is updated by setting the corresponding bit for the new miss (unless the previous miss is already the parent of a distance correlation miss).
- Searching for a distance-correlation parent starts after processing the first d misses, where d is the prefetch distance. The new miss can be recorded as a distance-correlation miss if the parent has empty free bits within the range for encoding the new miss. Note that a miss is recorded as a new distance-correlation miss only if the miss has not been recorded as either a bitmap or a distance-correlation miss previously.
- A single mode bit is used to distinguish the two types of miss correlations.

When a L2 block is replaced, the block is compressed before writeback to DRAM. Depending on the number of free bits f from the compression algorithm, the encoding of

the miss correlation works as follows. Each DRAM block has one extra bit c to indicate whether the block is compressed. The block is compressed, i.e. c=1, if $f \geq h$, where h is a pre-set fixed bits for the miss correlation. Otherwise, the block is not compressed and no prefetching information is encoded. When $f \geq h$, the miss correlations along with a mode bit m are concatenated with the compressed data block and sent to DRAM.

Figure 4-7 illustrates a simple example of establishing the per-block correlation. In this example, a sequence of misses, A, A+1, A-2, A+3, B, B+1, A+2, and C, is recorded in the MHB from the bottom to the top, where $A \pm i$, $B \pm j$, and $C \pm k$ represent block addresses in three distinct regions. Blocks in A region are reachable from blocks in B region and vice verse using the distance correlations and blocks in C region are unreachable from blocks in either A or B. For simplicity, we assume that only 4 correlation bits record ± 2 blocks from the base block and the prefetch distance is 2. A single mode bit indicates either the bitmap (b) or the distance (d) correlation is established.

The first miss A enters an empty MHB. The second miss A+1 is a neighbor of A. The mode bit (b) and the bitmap correlation (0010) of A are updated. The third A-2 and fourth A+3 misses are also encoded as a correlated neighbor in A and A+1. Miss B is not a neighbor of any previous miss in the MHB. The search for the distance-based correlation starts from A-2 since the prefetch distance is 2. A-2 is chosen as the parent because it is not a correlated parent of any other miss. Miss A+2 is recorded as a correlated neighbor for A, A+1 and A+3. When miss C comes, it cannot be recorded since it is not a neighbor of any previous miss, nor can be recorded using the distance correlation due to insufficient correlation bits. When A is removed from MHB, the

correlation bits (1011) are saved in the cache and eventually written back to DRAM. When A is missed again, the correlation bits trigger the prefetch of A-2, A+1, and A+2.

4.6 Performance Results

We simulate and compare performance of four L2 prefetching methods including a regular stream/stride prefetcher (STREAM) [58, 60], a Sampled Temporary Memory Streaming prefetcher (STMS) [55], a Spatio-Temporal Memory Streaming prefetcher (STeMS) [56], and the proposed per-block miss correlation prefetcher (PBMC). All STMS, STeMS and PBMC include an integrated stream/stride prefetcher to handle regular streaming misses. The simulation methodology is described in Section 1.4, parameters for the simulator are listed in Table 1-2. For each scheme evaluated, specific parameters are given in Table 4-2.

In STREAM, we record 16 streams per core. The prefetch distance is 32 with prefetch degree of 4. Constant stride prefetching is also integrated to prefetch non-consecutive blocks. For stream-enhanced STMS, STeMS and PBMC prefetchers, the regular stream/stride prefetching takes precedence over other correlation prefetching schemes.

In the STMS prefetcher, we simulate an off-chip 32MB, 12-way index table with a history buffer of 64MB and other parameters follow the design in [67]. The prefetches and the metadata fetch/lookup requests go through a 96-entry prefetch queue. The index table and history buffer updates go through a separate 64-entry writeback queue. The demand misses in the MSHR have the highest priority over the prefetch requests in the prefetch queue, while the metadata updates have the lowest priority. For STeMS, we use a 32MB index table to identify the most recent occurrence of the miss in the RMOB. To predict the repeated spatial access patterns, a 16K-entry pattern history

table is implemented with a 32-entry filter table and a 64-entry accumulation table in each core. Other parameters are the same as in [56].

For PBMC, we use a 128-entry miss history buffer to decide the time correlation. For space correlation, we simulate 15 free bits per block for encoding the miss correlation. The prefetch distance is 16 for encoding the distance-based correlation. The priority of handling demand misses, the prefetch requests, and the miss correlation updates is handled the same way as in STMS. In simulating the pipeline execution, compression and decompression take 10 and 4 cycles respectively.

This section provides the performance evaluation of the four prefetching schemes. First, we compare the IPC improvement among the four schemes. Second, we studied the prefetch accuracy, miss coverage and extra traffic for each prefetcher. Third, we show the evaluation of the bitmap-only, distance-only and bitmap plus distance implementations of PBMC prefetchers. Lastly, we show the sensitivity studies of PBMC with respect to the size of the MHB and the number of per-block free bits. All results are normalized to the performance without prefetching.

4.6.1 IPC Comparison

Figure 4-8 summarizes the normalized IPC improvements for the four prefetching methods. The overall IPCs of the 10 workloads using the PBMC prefetching method are given in Table 4-3. PMBC provides the best IPC improvement in nine out of ten workloads. Among the workloads, PBMC significantly outperforms other perfetchers in radix, lbm and health. PBMC as well as others provide significant IPC improvement over no-prefetcher in libquantum, stream, swim, equake, ocean and facesim. We will explain the performance of each workload in the rest of this section.

For radix, PBMC achieves 138% improvement over no-prefetcher while STREAM, STMS and STeMS provide only 34%, 19% and 20% improvement respectively. The implementation of radix has two distinct streaming behaviors for reading and writing. While the read stream is sequential, the write stream is made up of a number of small streams that starts randomly. The read stream can easily be handled by the stream prefetcher. The randomness in the write stream completely messed up the index table in STMS and STeMS prefetchers tricking them to think each random starting point as a new stream. This randomness however does not confuse PBMC as the miss-correlation is built for the output stream. No matter where the new starting point is within the stream, PBMC will accurately prefetch the subsequent blocks. This is the strength of the PBMC prefecher.

For lbm, the IPC improvement for PBMC is 63% while only 1%, 17% and 19% for the other three schemes. lbm is a bandwidth intensive workload where it reads from 19 streams and then updates 19 streams for the next iteration. The close to 40 streams is beyond the regular stream prefetcher can handle. It also overloaded the index table in STMS and STeMS. PBMC, on the other hand, provides unbounded per-block triggering points to handle the massive short streams. In addition, the high bandwidth demand of lbm causes delays in accessing the metadata from off-chip making STMS and STeMS less effective.

For health, PBMC provides 44% IPC improvement while the other three prefetchers have little or no improvement. As shown in Figure 4-2, multi-copy version of health spent significant amount of time in high MPKI regions which tends to saturate the

memory bus. The extra off-chip metadata accesses create additional traffic and degrade the overall performance.

For the stream workload, STMS, STeMS and PBMC perform significantly better than STREAM. Note that the stream we used is modified from original stream benchmark [39] to access irregular memory sequence. The extra metadata traffic degrades the performance of STMS and STeMS. PBMC shows the best performance with 104% improvement over no prefetching.

For libquantum, sph_hea, and equake, STMS and STeMS only perform marginally better than STREAM; while PBMC outperforms all three schemes. The main reason is that these workloads require high memory bandwidth and the off-chip prefetch metadata competes with the demand misses for the limited bandwidth. This is not a problem for PBMC since the metadata is embedded in the data blocks. Swim, ocean and facesim are dominated by the regular stream/stride behavior, hence all prefetchers have similar performance.

4.6.2 Prefetch Accuracy, Miss Coverage and Extra Traffic

The total memory traffic for the four prefetching schemes is plotted in Figure 4-9 and 4-10. Figure 4-9 shows the fetch traffic including the misses, the prefetch partial hits, the prefetch hits, the useless prefetches, and the fetches of the miss correlation metadata. The partial hits represent the condition that a demand miss occurs while the target block is being prefetched. Substantial traffic is generated in accessing the miss-stream metadata in running libquantum, lealth, sph_hea, radix, lbm, stream and equake using STMS and STeMS. Between the two, STeMS is less accurate with more useless prefetches especially in stream. These two prefetching methods also exhibit similar miss coverage. Among all the schemes, PBMC demonstrates the highest miss

coverage with high accuracy and little extra traffic. STREAM is also very accurate but it has the worst miss coverage. These traffic results are consistent with the IPC improvement in Figure 4-8.

Figure 4-10 displays the writeback traffic including the dirty-line writebacks, the clean-line writebacks for updating the miss correlation in PBMC, and the metadata updates needed in STMS and STeMS. A majority of the writeback traffic is for normal dirty-line writebacks. STMS and STeMS generate a significant amount of metadata updates in health, sph_hea, radix, lbm, and equake even with only 12.5% of probability-based index table updates. Most of the workloads have very small amount of clean-line writebacks under PBMC except for equake and facesim.

But when the memory bandwidth is saturated, the clean-line writebacks can hurt performance. In order to minimize the impact, we implemented a mechanism on writeback queue to simply drop the clean-line writebacks when the writeback queue is full. Therefore, the clean-line writebacks only happen when the memory bandwidth is available, and it has lower priority than demand misses and prefetches when competing for memory bandwidth.

4.6.3 PBMC Encoding Options

In Figure 4-11, we compare the IPC improvement for three miss-correlation encoding mechanisms in *PBMC* including bitmap-only, distance-only, and a combination of both. The results show that encoding distance-only miss correlation performs reasonable well, while encoding only the bitmap correlations performs poorly. Overall, the average IPC improvements are 36%, 65%, and 68% respectively for the bitmap-only, the distance-only and the combined *PBMC* prefetchers. Note that for

distance-only, there is no need for a mode bit, thus can double the distance in encoding the correlation in comparison with the distance encoding in the combined method.

4.6.4 PBMC Sensitivity to MHB Size and Number of Encoding Bits

We carry out sensitivity studies on two key PBMC design parameters, the MHB size and the number of bits for encoding the miss correlation as shown in Figure 4-12. We simulate the MHB size of 64, 128, 256, and 512. The results show that the IPC improvement is insensitive to these MHB sizes. In order to understand the performance impact on the number of bits for encoding the miss correlation, we collect the average IPCs for the 10 workloads using the PBMC prefetching method. The resulting average IPCs are 4.8, 5.2, 5.1, 5.0, and 5.0 respectively for using 12, 15, 18, 21, and 24 encoding bits, where the best performance is given to 15 encoding bits. Having more bits to encoding more correlations comes with a price with less memory blocks that can be used to encode the miss correlation. This is due to the lower compression coverage in producing the required bits.

4.7 Related Work

Many hardware data prefetching methods either regularity-based [30, 12, 24, 31, 42, 58, 60] or correlation-based [13, 65, 55, 14, 66, 56, 67] have been reported. The proposed prefetching method integrates a regular stream/stride prefetcher along with the space-time per-block miss-correlation prefetcher. The regular stream prefetcher [58, 60] detected missing address stream in either ascending or descending order to prefetch consecutive blocks in the same direction. An integrated stride prefetching [12, 24] can improve the stream prefetching to avoid fetching consecutive but useless blocks.

In the original pair-wise miss-correlation prefetcher [9], each miss records its subsequent miss as the correlated miss for prefetching. The Markov [29] prefetcher

generalizes the pair-wise prefetcher by allowing each miss to have multiple children misses. Based on the observed probabilities of the correlated children, prefetches can be issued for one or more children. The proposed per-block space-time miss-correlation prefetcher further generalizes the miss correlation by relaxing the adjacency of misses to space and time proximity for defining the correlated misses.

Cache misses in the form of temporal repeated streaming have been reported [13]. By recording the stream history, the blocks in each stream can be prefetched together. Memory accesses also exhibit repetitive layouts in large fixed memory regions as reported in the spatial streaming prefetchers [55, 56]. These repeated spatial patterns are predictable using PC-based correlation along with the region offset. Spatio-Temporal memory streaming [56] exploits both spatial and temporal streaming patterns. It records and replays the temporal sequence of region accesses to reconstruct the total miss order. Similar idea is also reported in [18] to link various local streams into a predictable chains of missing streams. However, in these approaches, they need to maintain a huge correlated miss history and incurs significant delay and extra bandwidth requirement.

The Global History Buffer [43] proposes a general FIFO structure for recording and identifying nearby missing address patterns. In our approach, we implement a similar FIFO Miss History Buffer (MHB) to detect nearby correlated misses within the time and space constraint. There are two other approaches to reduce the space overhead for correlated miss prefetchers, Tag-Correlating prefetcher [27] extends the miss correlation to much bigger blocks. Coterminous Group prefetcher [53] records and prefetches only the nearby missing blocks with equal reuse distance.

There has been much research done on using data compression to expand the effective cache or memory capacity [68, 76, 61, 71, 20, 72, 2, 3]. IBM has commercialized the first compressed memory subsystem MXT [1, 61] which uses specialized hardware to compress and decompress the entire memory. As reported in performance study of IBM MXT [1], the primary goal of a compressed memory is saving in memory cost instead of performance improvement. According to [1], although the main memory can be compressed by a factor of 2.3 in average, IBM MXT improves only 1.3% over a standard system without compression based on evaluations on SPEC2000 CPU benchmark. Therefore compression on the entire main memory does not have much performance benefit .

Other improvements have been reported in [76, 71, 20]. The frequent value locality theory is also applied by Alameldeen and Wood [3] and Adl-Tabatabai et al. [2] for compressed cache design. Similarly, Yang's PBPM compression scheme [72] and the SVDE [47] compression scheme are both based on the frequent value locality. Both PBPM and SVDE schemes are based on dynamic dictionary to target on-line compression that requires low latency. The difference is that PBPM uses a small dictionary (16 entries) that resides in dedicated buffer to compress memory blocks; whereas the SVDE scheme uses only one word dictionary and the word is embedded in the compressed block.

A few researchers studied the interactions between data compression and data prefetching. Alameldeen and Wood discusses how the extra tags used in the compressed cache design can be used for prefetching in [3]. In [2], Adl-Tabatabai et al. shows using companion line prefetches can improve performance by 1.5x for workloads

running on their cache compression design. These earlier works use prefetches as a mechanism to improve performance for their compressed cache designs. This is in sharp contrast to our work where we use data compression to enable data prefetching. Closest to our work of using data compression for data prefetching is the work by Vitter, Krishnan and Curewitz in [63, 17] where they developed prefetching algorithms based on a probabilistic Markov model using the Lempel-Ziv compression algorithms [78, 79].

4.8 Summary

Previous correlation-based prefetchers record a long miss history to trigger prefetches incurring significant space overhead. Recent solutions store the streams of miss addresses in off-chip DRAMs, but the latency and memory bandwidth requirement for extra off-chip accesses of the metadata can degrade the benefit of data prefetching. In this chapter, we present a per-block miss correlation prefetcher which records the miss correlation in each DRAM block to hold an unbounded miss history. Per-block miss correlations can be detected and recorded dynamically based on the closeness in time and space among misses. To avoid space and bandwidth overhead, data compression techniques are employed to create free bits in each DRAM block for encoding the miss correlations. The encoded miss correlation can then be fetched along with the content of the block to avoid extra memory accesses. We evaluated the effectiveness of the proposed prefetcher using data-parallel workloads. The results demonstrate that the per-block miss correlation prefetcher has superior performance improvement over the existing stream/stride, temporal memory streaming, and spatial-temporal memory streaming prefetchers while maintaining minimum space overhead.

Table 4-1. The compression coverage of SVDE scheme on parallel workloads

	libquantum	health	sph_hea	radix	lbm	stream	swim	equake	ocean	facesim
Coverage	100%	100%	99.7%	99.9%	100%	100%	99.6%	41.1%	95.9%	11.9%

Table 4-2. Simulation parameters

Prefetcher	Configuration
STREAM	Number of streams (per core): 16, prefetch distance/degree: 32/4
STMS	Index table: 32MB 12-way, history circular buffer: 64MB, index table cache: 8KB, Index table update percentage: 12.5%
STeMS	Index table: 32MB 12-way, AGT: 64 entries, PST: 16K entries, RMOB: 128K entries, reconstruction buffer: 256 entries, SVB: 64 entries
PBMC	Miss history buffer (per-core): 128 entries, prefetch distance: 16, miss-correlation bits: 15 per block, compression: 10 cycles, decompression: 4 cycles

Table 4-3. The overall IPC

	radix	lbm	sph_hea	equake	libquantum	facesim	ocean	swim	stream	health
IPC	10.4	9.3	7.4	5.9	5.5	4.3	4.3	2.1	1.9	0.6

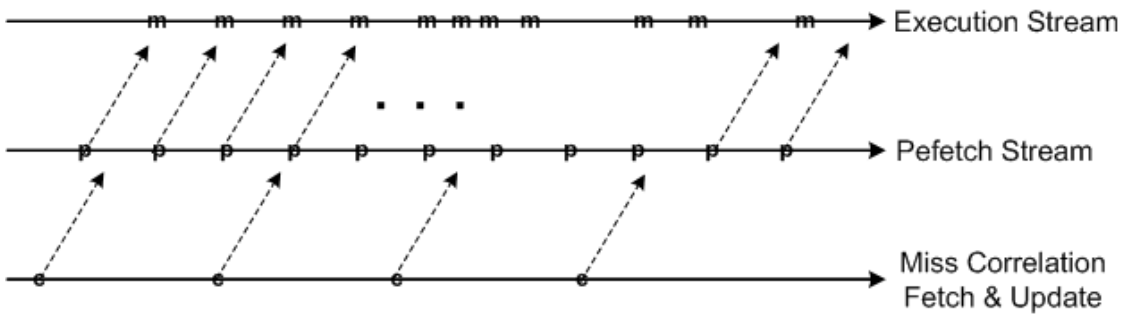

Figure 4-1. Timely and Seamless Prefetching

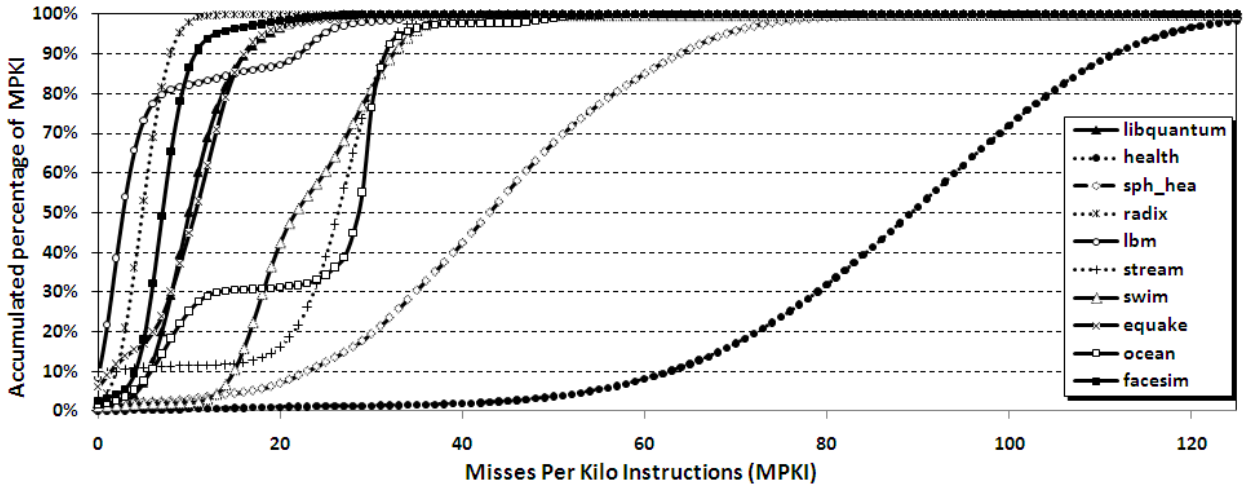

Figure 4-2. Cumulative distribution of MPKI for 10 parallel workloads running on 8-core CMP with 4MB shared L2 and 64-byte blocks. Data are periodically sampled from 10B instructions runs with the first 1B instructions as warm-up.

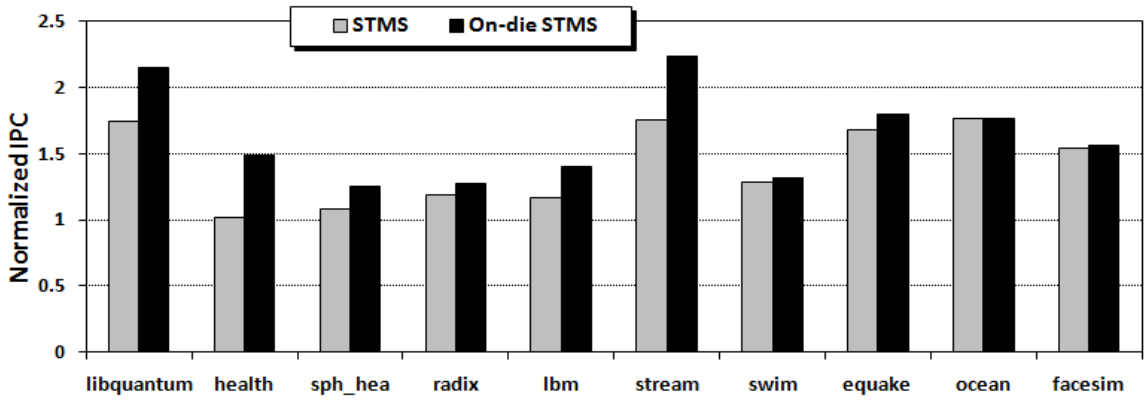

Figure 4-3. IPC comparison between STMS and on-die STMS prefetchers

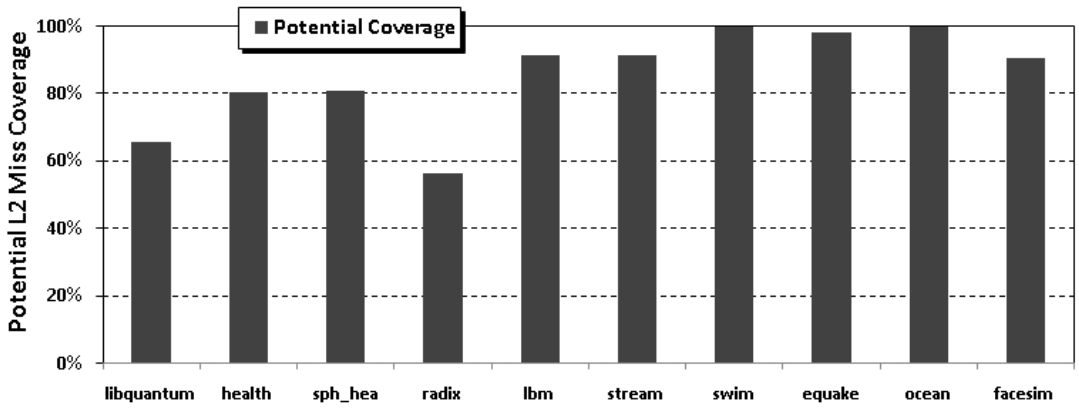

Figure 4-4. Potential miss coverage using per-block miss correlation

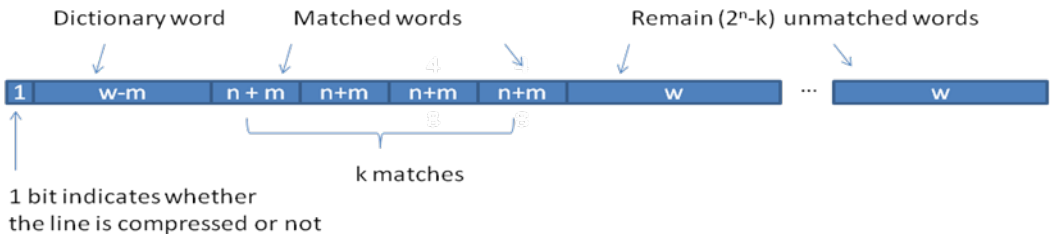

Figure 4-5. SVDE compressed block format

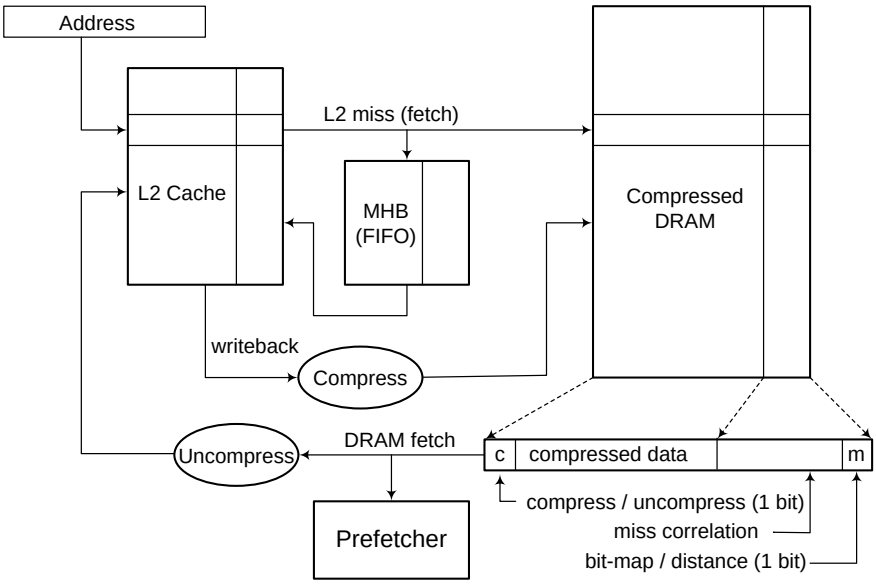

Figure 4-6. Basic architecture design of encoding per-block miss correlation in DRAM for prefetching

Block	Encoded Correlation	Mode
. . .		
C		
A+2		
B+1		
B	0010 (B+1)	b
A+3	0100 (A+2)	b
A-2	B	d
A+1	0001 (A+3)→ 0011 (A+2)	b
A	0010 (A+1)→1010 (A-2)→1011 (A+2)	b

MHB (FIFO)

Figure 4-7. An example of establishing per-block miss correlation

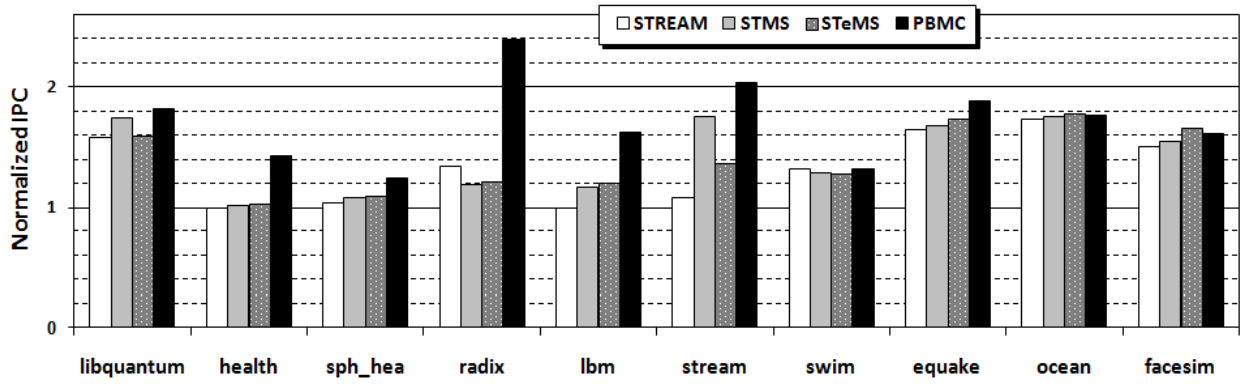

Figure 4-8. IPC improvement for 10 data-parallel workloads

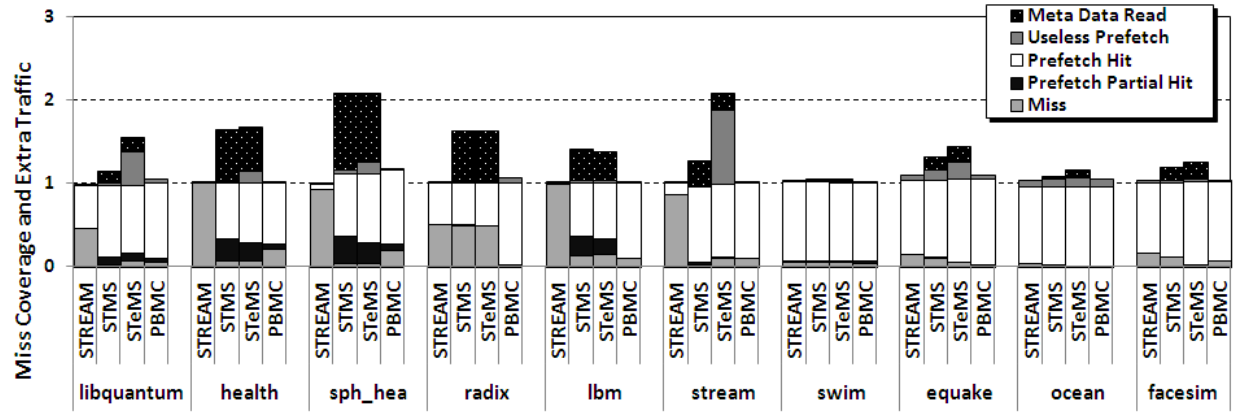

Figure 4-9. Accuracy, traffic, and miss coverage

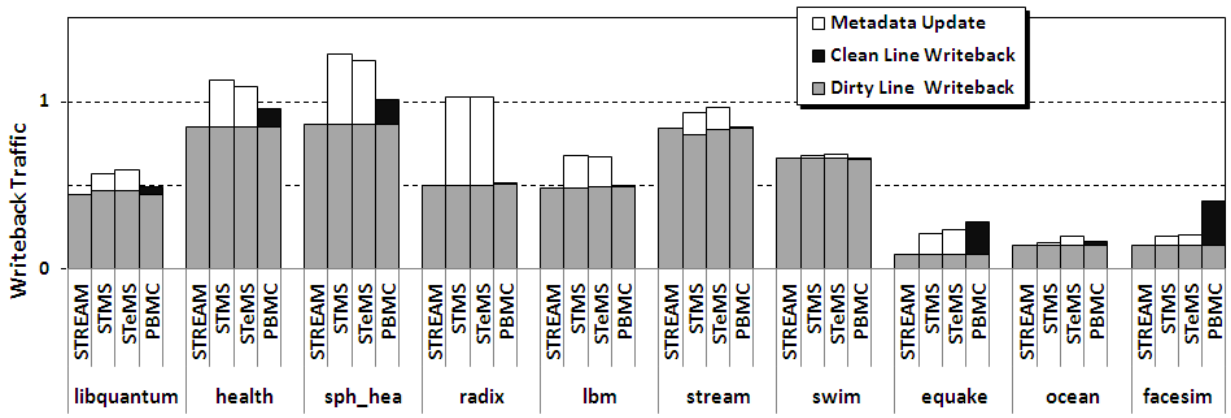

Figure 4-10. Writeback traffic

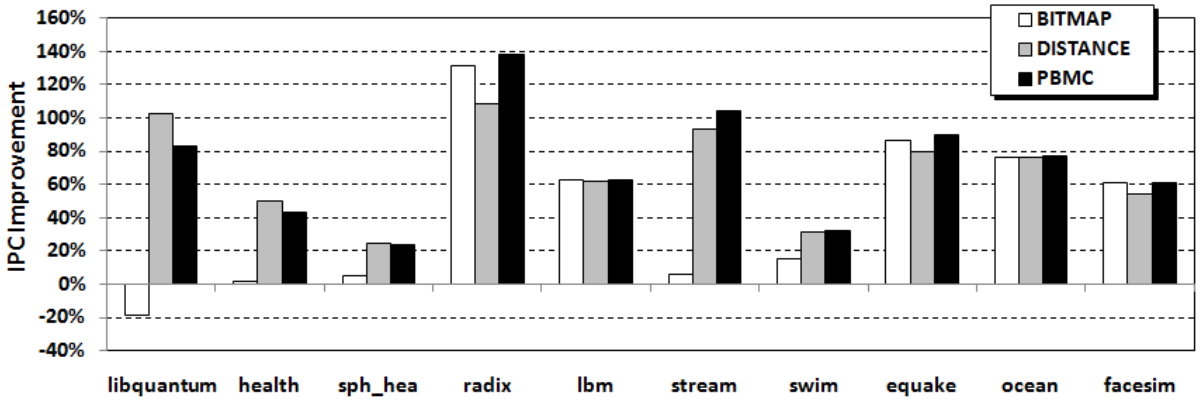

Figure 4-11. IPC improvement: Bitmap vs. Distance vs. combined PBMC

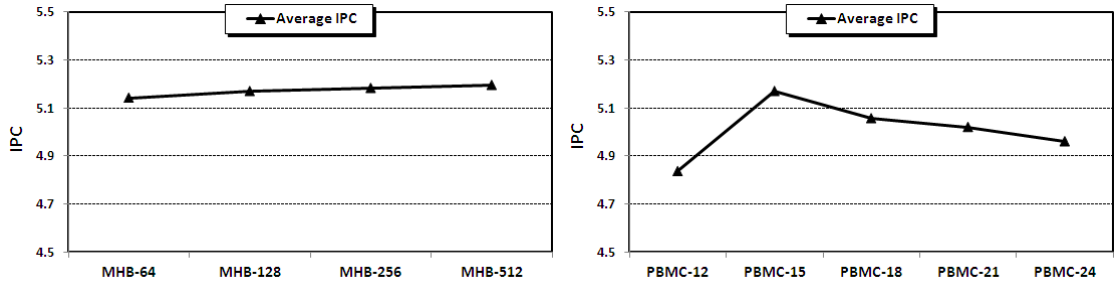

Figure 4-12. Sensitivity studies on MHB size and PBMC encoding bits

CHAPTER 5
DISSERTATION CONCLUSION

Based on three typical cache miss patterns introduced in Chapter 1, i.e., regular stream, linked data structure, and correlated misses, we present three works to prefetch these cache miss patterns in an accurate and timely way. These prefetching techniques require very little hardware cost, thus can be applied to traditional single-core processor as well as state-of-the-art CMPs. And evaluations have shown they can improve system performance significantly. With the ever-increasing gap between processor and memory, these prefetching techniques become more valuable. In the first work, we present an enhanced stream prefetcher, which takes a modern stream prefetcher design as base, and improves performance with several enhancement techniques. In the second work, we describe a semantics-aware prefetcher for accurate and timely LDS prefetching. The semantics-aware prefetcher dynamically establishes node structure of LDS through simple experimental rules based on the LDS traversal history. We also introduce three leap prefetching techniques to improve the lateness problem existing on earlier works. The third work introduces a novel prefetching mechanism to handle correlated misses. In this work, we encode per-block spatial/temporal miss correlations in compressed DRAM to provide unbounded miss history with minimal space and bandwidth overheads. These prefetching mechanisms have been verified by simulation results that they are effective on prefetching the three cache miss patterns as shown in Figure 1-2 with minimal hardware overhead.

LIST OF REFERENCES

[1] B. Abali, H. Franke, X. Shen et al., “Performance of Hardware Compressed Main Memory,” in 7th HPCA, 2001.

[2] A.-R. Adl-Tabatabai, A. M. Ghuloum, and S. O. Kanaujia, “Compression in cache design,” in 21st ACM International Conference on Supercomputing, 2007.

[3] A. R. Alameldeen, and D. A. Wood, “Interactions between compression and prefetching in chip multiprocessors,” in 13th HPCA, 2007.

[4] B. M. Beckmann, M. R. Marty, and D. A. Wood, “ASR: Adaptive Selective Replication for CMP Caches,” in 39th MICRO, 2006.

[5] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, vol. 5, no. 2, pp. 78-101, 1966.

[6] Olden Benchmark, http://www.cs.princeton.edu/~mcc/olden.html.

[7] SPEC 2006 benchmark, http://www.spec.org/cpu2006/.

[8] C. Bienia, S. Kumar, J. Singh et al., “The PARSEC benchmark suite: Characterization and architectural implications,” in 17th International Conference on Parallel Architectures and Compilation Techniques, 2008.

[9] M. Chamey, and A. Reeves, Generalized correlation based hardware prefetching, Technical Report EE-CEG-95-1, Cornell University, 1995.

[10] The 1st JILP Data Prefetching Championship (DPC-1), http://www.jilp.org/dpc/.

[11] J. Chang, and G. S. Sohi, “Cooperative cache partitioning for chip multiprocessors,” in 21st ICS, 2007.

[12] T.-F. Chen, and J.-L. Baer, “Reducing memory latency via non-blocking and prefetching caches,” in 5th ASPLOS, 1992.

[13] T. M. Chilimbi, and M. Hirzel, “Dynamic hot data stream prefetching for general-purpose programs,” in PLDI, 2002.

[14] Y. Chou, “Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications,” in 40th Micro, 2007.

[15] J. D. Collins, D. M. Tullsen, H. Wang et al., “Dynamic speculative precomputation,” in 34th MICRO, 2001.

[16] R. Cooksey, S. Jourdan, and D. Grunwald, “A stateless, content-directed data prefetching mechanism,” in 10th ASPLOS, 2002.

[17] K. Curewitz, P. Krishnan, and J. Vitter, “Practical prefetching via data compression,” ACM SIGMOD Record, vol. 22, no. 2, pp. 266, 1993.

[18] P. Diaz, and M. Cintra, “Stream chaining: Exploiting multiple levels of correlation in data prefetching,” in 36th ISCA, 2009.

[19] E. Ebrahimi, O. Mutlu, and Y. N. Patt, “Techniques for bandwidth-efficient prefetching of Linked Data Structures in hybrid prefetching systems,” in 15th HPCA, 2009.

[20] M. Ekman, and P. Stenstrom, “A Robust Main-Memory Compression Scheme,” in 32nd ISCA, 2005.

[21] S. Eyerman, and L. Ecckhout, “A Memory-Level Parallelism Aware Fetch Policy for SMT Processors,” in 13th HPCA, 2007.

[22] M. Ferdman, and B. Falsafi, “Last-touch correlated data streaming,” in ISPASS, 2007.

[23] FeS2: A Full-system Execution-driven Simulator for x86, http://fes2.cs.uiuc.edu/.

[24] J. Fu, J. H. Patel, and B. L. Janssens, “Stride directed prefetching in scalar processors,” in 25th MICRO, 1992.

[25] J. Hennessy, D. Patterson, D. Goldberg et al., Computer architecture: a quantitative approach: Morgan Kaufmann, 2003.

[26] L. R. Hsu, S. K. Reinhardt, R. Iyer et al., “Communist, utilitarian, and capitalist cache policies on CMPs: Caches as a shared resource,” in 15th PACT, 2006.

[27] Z. Hu, M. Martonosi, and S. Kaxiras, “TCP: Tag Correlating Prefetchers,” in 9th HPCA, 2003.

[28] J. Huh, D. Burger, and S. W. Keckler, “Exploring the design space of future CMPs,” in PACT, 2001.

[29] D. Joseph, and D. Grunwald, “Prefetching using Markov predictors,” in 24th ISCA, 1997.

[30] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in 17th ISCA, 1990.

[31] G. B. Kandiraju, and A. Sivasubramaniam, “Going the distance for TLB prefetching: An application-driven study,” in 29th ISCA, 2002.

[32] N. Kohout, S. Choi, D. Kim et al., “Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes,” in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Sep 2001.

[33] J.-S. Lee, W.-K. Hong, and S.-D. Kim, “Design and evaluation of a selective compressed memory system,” in IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1999.

[34] V. Lee, C. Kim, J. Chhugani et al., “Debunking the 100x GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU,” in 37th ISCA, 2010.

[35] H. Lieberman, and C. Hewitt, “A real-time garbage collector based on the lifetimes of objects,” Communications of the ACM, vol. 26, no. 6, pp. 429, 1983.

[36] C.-K. Luk, and T. C. Mowry, “Compiler-based prefetching for recursive data structures,” in 7th ASPLOS, 1996.

[37] P. Magnusson, M. Christensson, J. Eskilson et al., “Simics: A full system simulation platform,” Computer, pp. 50-58, 2002.

[38] M. Martin, D. Sorin, B. Beckmann et al., “Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset,” ACM SIGARCH Computer Architecture News, vol. 33, no. 4, pp. 92-99, 2005.

[39] McCalpin, and J. D, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” IEEE Computer Society Technical Committee on Computer Architecture Newsletter, 1995.

[40] N. Megiddo, and D. S. Modha, “Outperforming LRU with an adaptive replacement cache algorithm,” Computer, vol. 37, no. Compendex, pp. 58-65, 2004.

[41] T. C. Mowry, M. S. Lam, and A. Gupta, “Design and evaluation of a compiler algorithm for prefetching,” in 5th ASPLOS, 1992.

[42] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, “AC/DC: An adaptive data cache prefetcher,” in 13th PACT, 2004.

[43] K. J. Nesbit, and J. E. Smith, “Data cache prefetching using a global history buffer,” in Proceedings of 10th International Symposium on High Performance Computer Architecture, Febrary 2004.

[44] Y. Qian, D. d'Humieres, and P. Lallemand, “Lattice BGK models for Navier-Stokes equation,” EPL (Europhysics Letters), vol. 17, pp. 479, 1992.

[45] M. K. Qureshi, D. N. Lynch, O. Mutlu et al., “A Case for MLP-Aware Cache Replacement,” in 33rd ISCA, 2006.

[46] K. Rajan, and R. Govindarajan, “Emulating Optimal Replacement with a Shepherd Cache,” in 40th Micro, 2007.

[47] Internal Report, “Single-Value Dynamic Encoding,” Note that the authors' names are withheld to avoid revealing the identity of this submitted paper, 2010.

[48] A. Rogers, M. Carlisle, J. Reppy et al., “Supporting dynamic data structures on distributed-memory machines,” ACM Transactions on Programming Languages and Systems, vol. 17, no. 2, pp. 233-263, 1995.

[49] A. Roth, A. Moshovos, and G. S. Sohi, “Dependence based prefetching for linked data structures,” in 8th ASPLOS, 1998.

[50] A. Roth, and G. Sohi, “Effective jump-pointer prefetching for linked data structures,” in 26th ISCA, 1999.

[51] N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” in IPDPS, 2009.

[52] T. Sherwood, E. Perelman, G. Hamerly et al., “Automatically characterizing large scale program behavior,” in 10th ASPLOS, 2002.

[53] X. Shi, Z. Yang, J.-K. Peir et al., “Coterminous locality and coterminous group data prefetching on chip-multiprocessors,” in 20th IPDPS, 2006.

[54] Y. Solihin, J. Lee, and J. Torrellas, “Using a user-level memory thread for correlation prefetching,” in 29th ISCA, 2002.

[55] S. Somogyi, T. F. Wenisch, A. Ailamaki et al., “Spatial memory streaming,” in 33rd ISCA, 2006.

[56] S. Somogyi, T. F. Wenisch, A. Ailamaki et al., “Spatio-temporal memory streaming,” in 36th ISCA, 2009.

[57] SPEC benchmark, http://www.spec.org/.

[58] S. Srinath, O. Mutlu, H. Kim et al., “Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers,” in Proceedings of 13th International Symposium on High Performance Computer Architecture, Febrary 2007.

[59] V. Srinivasan, E. Davidson, and G. Tyson, “A prefetch taxonomy,” IEEE Transactions on Computers, vol. 53, no. 2, pp. 126-140, 2004.

[60] J. Tendler, S. Dodson, S. Fields et al., “POWER4 system microarchitecture,” IBM Technical White Paper, Oct 2001.

[61] R. Tremaine, P. Franaszek, J. Robinson et al., “IBM memory expansion technology (MXT),” IBM Journal of Research and Development, vol. 45, no. 2, pp. 271-285, 2001.

[62] J. Tuck, L. Ceze, and J. Torrellas, “Scalable Cache Miss Handling for High Memory-Level Parallelism,” in 39th Micro, 2006.

[63] J. S. Vitter, and P. Krishnan, “Optimal prefetching via data compression,” in 32nd Annual Symposium on Foundations of Computer Science, 1991.

[64] Z. Wang, D. Burger, K. S. McKinley et al., “Guided region prefetching: a cooperative hardware/software approach,” in 30th ISCA, 2003.

[65] T. F. Wenisch, S. Somogyi, N. Hardavellas et al., “Temporal Streaming of Shared Memory,” in 32nd ISCA, 2005.

[66] T. F. Wenisch, M. Ferdman, A. Ailamaki et al., “Temporal streams in commercial server applications,” in IEEE International Symposium on Workload Characterization, 2008.

[67] T. F. Wenisch, M. Ferdman, A. Ailamaki et al., “Practical off-chip meta-data for temporal memory streaming,” in 15th HPCA, 2009.

[68] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, “The case for compressed caching in virtual memory systems,” in Proceedings of USENIX Annual Technical Conference, 1999.

[69] S. C. Woo, M. Ohara, E. Torrie et al., “SPLASH-2 programs: Characterization and methodological considerations,” in 22nd ISCA, 1995.

[70] W. Wulf, and S. McKee, “Hitting the memory wall: Implications of the obvious,” ACM SIGARCH Computer Architecture News, vol. 23, no. 1, pp. 20-24, 1995.

[71] J. Yang, and R. Gupta, “Frequent value locality and its applications,” ACM Trans. on Embedded Computing Systems, vol. 1, no. 1, pp. 79-105, 2002.

[72] L. Yang, H. Lekatsas, and R. P. Dick, “High-performance operating system controlled memory compression,” in 43rd ACM/IEEE Design Automation Conference, 2006.

[73] M. T. Yourst, “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator,” in ISPASS, 2007.

[74] C. Yuan, B. Fahs, and S. Abraham, “Microarchitecture optimizations for exploiting memory-level parallelism,” in 31st ISCA, 2004.

[75] C. Yuan, L. Spracklen, and S. G. Abraham, “Store memory-level parallelism optimizations for commercial applications,” in 38th Micro, 2005.

[76] Y. Zhang, J. Yang, and R. Gupta, “Frequent value locality and value-centric data cache design,” in 9th ASPLOS, 2000.

[77] Y. Zhang, and R. Gupta, “Enabling partial cache line prefetching through data compression,” in International Conference on Parallel Processing, 2003.

[78] J. Ziv, and A. Lempel, “A universal algorithm for sequential data compression,” IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, 1977.

[79] J. Ziv, and A. Lempel, “Compression of individual sequences via variable-rate coding,” IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, 1978.

BIOGRAPHICAL SKETCH

Gang Liu received his B.E. degree in Department of Computer Science and Engineering at Tsinghua University in 2000, and M.E. degree in the same major from Institute of Software, Chinese Academy of Sciences in 2003. After graduation, he worked as software engineer for a start-up company Baidu in Beijing. One year later, he started to pursue Ph.D. degree in Department of Computer and Information Science and Engineering at University of Florida. His research area is computer architecture.