Mechanisms to Improve the Efficiency of Hardware Data Prefetchers

Pedro Díaz

Doctor of Philosophy

School of Informatics

University of Edinburgh

2010

Abstract

A well known performance bottleneck in computer architecture is the so-called memory wall. This term refers to the huge disparity between on-chip and off-chip access latencies. Historically speaking, the operating frequency of processors has increased at a steady pace, while most past advances in memory technology have been in density, not speed. Nowadays, the trend for ever increasing processor operating frequencies has been replaced by an increasing number of CPU cores per chip. This will continue to exacerbate the memory wall problem, as several cores now have to compete for off-chip data access. As multi-core systems pack more and more cores, it is expected that the access latency as observed by each core will continue to increase. Although the causes of the memory wall have changed, it is, and will continue to be in the near future, a very significant challenge in terms of computer architecture design.

Prefetching has been an important technique to amortize the effect of the memory wall. With prefetching, data or instructions that are expected to be used in the near future are speculatively moved up in the memory hierarchy, were the access latency is smaller. This dissertation focuses on hardware data prefetching at the last cache level before memory (last level cache, LLC). Prefetching at the LLC usually offers the best performance increase, as this is where the disparity between hit and miss latencies is the largest.

Hardware prefetchers operate by examining the miss address stream generated by the cache and identifying patterns and correlations between the misses. Most prefetchers divide the global miss stream in several sub-streams, according to some pre-specified criteria. This process is known as localization. The benefits of localization are well established: it increases the accuracy of the predictions and helps filtering out spurious, non-predictable misses. However localization has one important drawback: since the misses are classified into different sub-streams, important chronological information is lost. A consequence of this is that most localizing prefetchers issue prefetches in an untimely manner, fetching data too far in advance. This behavior promotes data pollution in the cache.

The first part of this thesis proposes a new class of prefetchers based on the novel concept of Stream Chaining. With Stream Chaining, the prefetcher tries to reconstruct the chronological information lost in the process of localization, while at the same time keeping its benefits. We describe two novel Stream Chaining prefetching algorithms based on two state of the art localizing prefetchers: PC/DC and C/DC. We

show how both prefetchers issue prefetches in a more timely manner than their non-chaining counterparts, increasing performance by as much as 55% (10% on average) on a suite of sequential benchmarks, while consuming roughly the same amount of memory bandwidth.

In order to hide the effects of the memory wall, hardware prefetchers are usually configured to aggressively prefetch as much data as possible. However, a highly aggressive prefetcher can have negative effects on performance. Factors such as prefetching accuracy, cache pollution and memory bandwidth consumption have to be taken into account. This is specially important in the context of multi-core systems, where typically each core has its own prefetching engine and there is high competition for accessing memory. Several prefetch throttling and filtering mechanisms have been proposed to maximize the effect of prefetching in multi-core systems. The general strategy behind these heuristics is to promote prefetches that are more likely to be used and cause less interference. Traditionally these methods operate at the *source* level, i.e., directly into the prefetch engine they are assigned to control.

In multi-core systems all prefetches are aggregated in a FIFO-like data structure called the Prefetch Request Queue (PRQ), where they wait to be dispatched to memory. The second part of this thesis shows that a traditional FIFO PRQ does not promote a timely prefetching behavior and usually hinders part of the performance benefits achieved by throttling heuristics. We propose a novel approach to prefetch aggressiveness control in multi-cores that performs throttling at the PRQ (i.e., *global*) level, using global knowledge of the metrics of all prefetchers and information about the global state of the PRQ. To do this, we introduce the Resizable Prefetching Heap (RPH), a data structure modeled after a binary heap that promotes timely dispatch of prefetches as well as fairness in the distribution of prefetching bandwidth. The RPH is designed as a drop-in replacement of traditional FIFO PRQs. We compare our proposal against a state-of-the-art source-level throttling algorithm (HPAC) in a 8-core system. Unlike previous research, we evaluate both multiprogrammed and multithreaded (parallel) workloads, using a modern prefetching algorithm (C/DC). Our experimental results show that RPH-based throttling increases the throttling performance benefits obtained by HPAC by as much as 148% (53.8% average) in multiprogrammed workloads and as much as 237% (22.5% average) in parallel benchmarks, while consuming roughly the same amount of memory bandwidth. When comparing the speedup over fixed degree prefetching, RPH increased the average speedup of HPAC from 7.1% to 10.9% in multiprogrammed workloads, and from 5.1% to 7.9% in parallel benchmarks.

Acknowledgements

I would like to thank my supervisor Marcelo Cintra for all the help and guidance he has provided me during these years. I have learned many things from Marcelo but the two most important lessons I have taken from him are how to carry serious research and how to persevere when things do not work out as expected. I value the flexibility and patience that he has shown during my PhD, as well as the freedom he has given me to pursue my research interests. I am also greatly indebted to Marcelo for the opportunity to spend a six month internship at a top research lab such as Intel Labs Germany. Lastly, I am particularly grateful to him for his insightful corrections and constant attention during the writing of this thesis.

I would also like to thank everybody at Institute for Computer Systems Architecture for their support and help. I am thankful to my second supervisor Ariteides Efthymiou and Nigel Topham for being part of my annual thesis review panel. I would also like to thank Mike O'Boyle and Björn Franke for their role in creating a vibrating research group that I have always been pround being part of. My fellow PhD students have made this journey much more enjoyable and interesting. Many thanks to Salman Khan, Hugh Leather, Nikolas Ioannou, Konstantina Mitropoulou, Sofia Pediaditaki, Vasileios Porpodas, Georgios Tournavidis and Polychronis Xekalakis. I am specially grateful to Polychronis for the countless hours of stimulating research conversation by the coffee machine, and generally, for being a constant source of ideas and advice.

On a personal note, I am very grateful to Raniska for her understanding and moral support over these years. Finally, and most importantly, I would like thank my parents Pedro and Cristina for their absolutely unconditional support and encouragement during my PhD.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following paper:

- "Stream Chaining: Exploiting Multiple Levels of Correlation in Data Prefetching". Pedro Díaz and Marcelo Cintra. International Symposium on Computer Architecture 2009.

(*Pedro Díaz*)

Table of Contents

1 Introduction **1**

1.1 Motivation and Focus of this Dissertation 1

1.2 Main Contributions 3

1.2.1 Improving Prefetching Timeliness With Stream Chaining . . . 3

1.2.2 Prefetching in Multi-Core Systems with Resizable Prefetch Heaps 5

1.3 Thesis Structure 6

2 Background **8**

2.1 Memory Hierarchies 8

2.2 Prefetching 10

2.3 Basic Operation of a Hardware Data Prefetcher 11

2.4 Prefetching Metrics 12

2.5 Hardware Prefetching Methods 13

2.5.1 Localization 13

2.5.2 Correlation 15

2.5.3 Other Approaches to Data Prefetching 20

2.6 Hardware Structures for Data Prefetching 21

2.6.1 Miss History hardware data structures 21

2.6.2 The Prefetch Request Queue 23

2.7 Adaptive Prefetch Throttling and Filtering 24

2.7.1 Prefetch Throttling Techniques 24

2.7.2 Prefetch Filtering Techniques 25

2.8 Prefetching in Multi-Core Systems 26

3 Stream Chaining **28**

3.1 Introduction 28

3.2 Accuracy and Timeliness in Localizing Prefetchers 28
3.3 Stream Chaining 29
3.4 Miss Graph Prefetching 32
3.4.1 PC/DC/MG 33
3.4.2 C/DC/MG 40
3.4.3 Table-based Alternative Implementations of PC/DC/MG and C/DC/MG 49

4 Evaluation of Stream Chaining Prefetchers 53
4.1 Simulation Setup 53
4.2 Benchmarks 54
4.3 Benchmark Characterization 54
4.3.1 L2 Cache Size Sensitivity 54
4.3.2 Miss Distances 56
4.4 PC/DC/MG 57
4.4.1 Performance and Traffic 57
4.4.2 Coverage and Accuracy 63
4.4.3 PC Stream Prediction Accuracy 65
4.4.4 Miss Graph Characterization 66
4.5 C/DC/MG 70
4.5.1 Performance and Traffic 70
4.5.2 Coverage and Accuracy 73
4.5.3 CZone Transition Prediction Accuracy 75
4.5.4 CZone Repetition Prediction Accuracy 76
4.5.5 Miss Graph Characterization 78
4.6 Comparison of PC/DC/MG, C/DC/MG and G/DC 80

5 Resizable Prefetch Heaps 85
5.1 Introduction 85
5.2 Prefetch Throttling as Producer-Consumer Problem 86
5.3 Case Study: The Hierarchical Prefetch Aggressiveness Control 87
5.4 Resizable Prefetch Heaps 89
5.4.1 Introduction 89
5.4.2 Prefetch Throttling with RPHs 93

6 Evaluation of Resizable Prefetch Heaps **98**
6.1 Simulation Setup . . . 98
6.2 Benchmarks . . . 98
6.3 Prefetch Mechanism and Throttling Strategies . . . 99
6.4 Metrics . . . 101
6.4.1 Performance . . . 101
6.4.2 Traffic . . . 101
6.4.3 Prefetch Fairness . . . 101
6.5 Benchmark Characterization . . . 103
6.5.1 L2 Cache-Performance Sensitivity . . . 103
6.5.2 L2 Cache Hit Rates and Usage . . . 104
6.6 Prefetching Performance . . . 107
6.7 Bandwidth Usage . . . 109
6.8 Prefetch Fairness . . . 110
6.9 Influence of Adaptive Resizing . . . 111
6.10 Characterization RPH Queues . . . 113
6.10.1 Average Number of Comparisons per Queue Operation . . . 113
6.10.2 Scaling States Histogram . . . 115

7 Summary of Contributions and Concluding Remarks **117**
7.1 Summary of Contributions of this Thesis . . . 117
7.1.1 Future Work . . . 118
7.2 Concluding Remarks . . . 119

A Benchmark Descriptions **121**
A.1 SPEC CPU2006 benchmarks . . . 121
A.1.1 Integer Benchmarks. . . . 121
A.1.2 Floating Point Benchmarks . . . 123
A.2 BioBench Benchmarks . . . 124
A.3 Parallel Benchmarks . . . 125
A.3.1 PARSEC benchmarks . . . 126
A.3.2 ALP Benchmarks . . . 127
A.3.3 Standalone programs . . . 128

Bibliography **129**

List of Tables

4.1 Simulated architectural parameters. 54
4.2 PC/DC/MG: miss graphs statistics (a) and GHB hop counts (b). 69
4.3 C/DC/MG: miss graphs statistics (a) and GHB hop counts (b). 79

5.1 HPAC decision rules 88
5.2 Memory utilization scale used to resize the RPH. 93
5.3 RPH throttling actions. 96

6.1 Architectural parameters: Per core (left) and system-wide (right). 99
6.2 Multi-programmed workloads. 100
6.3 Example Prefetch Fairness calculation. 103
6.4 Scaling states histogram for multi-programmed workloads (left) and parallel benchmarks (right) 115

List of Figures

2.1 Standard memory hierarchy with typical access latencies. 9
2.2 Execution context, spatial and temporal localization. 15
2.3 Example of Address Correlation. 17
2.4 Example of Markov Prefetching. 18
2.5 Example of Constant Stride correlation. 19
2.6 Example of Delta correlation 20
2.7 Table structure for the PC/DC prefetcher 22
2.8 GHB for the PC/DC prefetcher. 23
2.9 (a) Overall logical organization of a multi-core architecture; (b) Logical organization of L2 prefetchers. 27

3.1 Example of Stream Chaining applied to PC localization: (a) global miss stream, (b) localized streams according to the PC of the missing instruction; and (c) one possible chaining of the localized streams. . . 31
3.2 (a) PC/DC/MG extensions to the GHB. (b) Resulting Miss Graph from the example miss stream. 35
3.3 PC/DC/MG Miss Graph examples from real applications: (a) *bzip2* with 2MB L2. (b) *dealII* with 256KB L2 (right). 36
3.4 PC/DC/MG Miss Graph cases: (a) non-cyclic chain longer than the prefetch degree (8); (b) cyclic chain shorter than the prefetch degree; and (c) non-cyclic chain shorter than the prefetch degree. 37
3.5 CZone transitioning problem: (a) global miss stream, PC localization and CZone localization; (b) Miss Graphs for PC and CZone localizations using the original PC/DC/MG graph construction algorithm. . . 41
3.6 (a) C/DC/MG extensions to the GHB. (b) Resulting Miss Graph from the example miss stream. 43

3.7 C/DC/MG Miss Graph examples from real applications: (a) *bzip2* with 2MB L2; (b) *dealII* with 256KB L2. 45
3.8 C/DC/MG Miss Graph cases. The numbers adjacent to each node show the CRC/LRC counters. (a) non-cyclic chain with prefetch potential bigger than or equal to the prefetch degree (16); (b) cyclic chain with lower prefetch potential than the prefetch degree; and (c) non-cyclic chain with lower prefetch potential than the prefetch degree. 45
3.9 Design of a table-based PC/DC prefetcher. 50
3.10 (a) PC/DC/MG extensions to the history table. (b) Resulting Miss Graph from the example miss stream. 51
3.11 (a) C/DC/MG extensions to the history table. (b) Resulting Miss Graph from the example miss stream. 52

4.1 L2 cache size sensitivity for 256KB, 512KB and 2MB L2 caches. . . 55
4.2 L2 cache Read Hit Rate for 256KB, 512KB and 2MB L2 caches. . . . 56
4.3 Number of L2 cache accesses per 1K instructions. 57
4.4 Miss distances for 256KB L2. Figure shows global (left), per PC (middle) and per CZone (right) miss distances. 59
4.5 Miss distances for 2MB L2. Figure shows global (left), per PC (middle) and per CZone (right) miss distances. 60
4.6 PC/DC (left bar) and PC/DC/MG (right bar) performance and traffic. 256KB L2 cache. 61
4.7 PC/DC (left bar) and PC/DC/MG (right bar) performance and traffic. 2MB L2 cache. 62
4.8 Prefetching coverage of PC/DC and PC/DC/MG. 256KB L2 cache. . . 64
4.9 Prefetching coverage of PC/DC and PC/DC/MG. 2MB L2 cache. . . . 65
4.10 Prefetching accuracy of PC/DC and PC/DC/MG. 256KB L2 cache. . . 66
4.11 Prefetching accuracy of PC/DC and PC/DC/MG. 2MB L2 cache. . . . 67
4.12 PC/DC/MG accuracy in predicting the next PC to appear in the next *w* misses. 256KB L2 Cache. 68
4.13 PC/DC/MG accuracy in predicting the next PC to appear in the next *w* misses. 2MB L2 Cache. 70
4.14 C/DC (left bar) and C/DC/MG (right bar) performance and traffic. 256KB L2 cache. 71

4.15 C/DC (left bar) and C/DC/MG (right bar) performance and traffic. 2MB L2 cache. 72
4.16 Prefetching coverage of C/DC and C/DC/MG. 256KB L2 cache. . . . 74
4.17 Prefetching coverage of C/DC and C/DC/MG. 2MB L2 cache. 75
4.18 Prefetching accuracy of C/DC and C/DC/MG. 256KB L2 cache. . . . 76
4.19 Prefetching accuracy of C/DC and C/DC/MG. 2MB L2 cache. 77
4.20 C/DC/MG accuracy in predicting the next CZone to appear in the next *w* misses. 256KB L2 Cache. 78
4.21 C/DC/MG accuracy in predicting the next CZone to appear in the next *w* misses. 2048KB L2 Cache. 80
4.22 CZone repetition accuracy. 256KB L2 cache. 81
4.23 CZone repetition accuracy. 2MB L2 cache. 82
4.24 Comparison of Stream Chaining prefetchers and G/DC. 256KB L2 cache. 83
4.25 Comparison of Stream Chaining prefetchers and G/DC. 2MB L2 cache. 84

5.1 Logical view of a binary heap: (a) as a tree; and (b) as an array. 90
5.2 Heap array split into priority and satellite data arrays. 91
5.3 Role of D and D_{limit} registers. W_n denotes the prefetch request wave number. 96

6.1 Cache sensitivity in multi-programmed workloads. 104
6.2 Cache sensitivity in parallel benchmarks. 105
6.3 L2 Read Hit Rate for multi-programmed workloads. 105
6.4 Number of accesses to the L2 cache per million cycles for multi-programmed workloads. 106
6.5 L2 Read Hit Rate for parallel benchmarks. 107
6.6 Number of accesses to the L2 cache per million cycles for parallel benchmarks. 107
6.7 Prefetching performance of multi-programmed workloads. 108
6.8 Prefetching performance of parallel benchmarks. 109
6.9 Prefetching bandwidth increase in multi-programmed workloads. . . . 110
6.10 Prefetching bandwidth increase in parallel benchmarks. 110
6.11 RPH throttling prefetch fairness normalized to HPAC prefetch fairness. 111
6.12 Results with and without adaptive resizing in multi-programmed workloads. 112
6.13 Results with and without adaptive resizing in parallel benchmarks. . . 112

6.14 Average number of comparisons per RPH insertion/extraction operation in multi-programmed workloads. 114
6.15 Average number of comparisons per RPH insertion/extraction operation in parallel benchmarks. 114

Chapter 1

Introduction

1.1 Motivation and Focus of this Dissertation

The term Memory Wall [15, 16] refers to the well known performance gap between processor speed and memory access latency. Nowadays off-chip memory requests have a latency of a few hundred processor cycles. Historically, technology has enabled a steady increase in processor operating frequencies while the most significant advances in memory technology have been related to density and not speed, thus creating an ever widening performance gap. At the present time the trend for increasing operating frequency in processors has flattened out. However, this trend has been exchanged for an increasing number of cores per chip. This has the consequence of more concurrent memory accesses and, therefore, an increase in the average off-chip access latency as observed by each core. As a result, even though the difference between processor and memory speed has not been increasing as steeply as in the past, the memory wall is expected to grow bigger.

Several techniques have been proposed to mask the big latencies related to off-chip memory access. Some try to exploit inherent Instruction Level Parallelism (ILP) in order to keep the processor busy with work while the required data arrives. Examples of these techniques include out-of-order execution, register renaming and speculative execution. The level of ILP varies, however, wildly across programs and even between program phases.

Prefetching is another technique historically successful in reducing the observed memory latency. In hardware prefetching, a set of hardware units observe the current memory access patterns. Based on this information, data that is likely to be needed in the future is fetched from memory and placed in one of the cache memory levels. In

software prefetching, the compiler uses static program analysis to interleave prefetch instructions throughout the program.

Both hardware and software prefetching have their benefits and drawbacks. In software prefetching there are no timing constraints for the execution of the prefetching algorithm, allowing the application of sophisticated prediction heuristics. However, software prefetching is mostly limited by static program analysis. On the other hand, hardware prefetching algorithms have access to the run-time information of the program, which is very valuable for predicting future accesses. Additionally, hardware prefetching is universally applicable (i.e., it is available to every program being run), whereas software prefetching requires recompilation or at least modification of the program binary. By contrast to software prefetching, hardware prefetching algorithms cannot be arbitrarily complex, and their run time has to fit into the timing constraints imposed to the hardware.

In this dissertation we will focus on hardware data prefetching into the lowest on-chip data cache level (typically the L2). This is in accordance to most recent research. There are three reasons for concentrating our study in this cache level:

- The miss latency on higher cache levels is usually quite small and processors can tolerate them without too much degradation in performance.
- The time constraints for implementing prefetching algorithms at the lowest level are much more lenient. This allows us to design more sophisticated algorithms that would be simply unrealistic to implement at a higher cache level
- Finally, the lowest on-chip cache level has the highest miss penalty, since data requests must use the much slower memory interface. Therefore it is on this level were a good, sophisticated prefetching algorithm is expected to provide the greatest returns on investment.

Although hardware prefetching has been shown to improve significantly the performance of the memory subsystem, it is not a technique without problems. An inaccurate prefetcher can generate copious amounts of wasteful memory traffic that will pollute the cache and produce, in the worst case, performance degradation compared to a configuration without prefetching. Simple hardware prefetchers can not capture the complex memory access patterns present in nowadays applications and therefore suffer from low coverage. On the other hand, many of the more complex prefetching algorithms proposed by previous research are too complicated or require too many

hardware resources to be implemented in current commercial architectures. While accurate prefetchers with good coverage are desirable, research on prefetching has traditionally focused almost exclusively on these two metrics, sidetracking the equally important aspect of improving prefetch timeliness (i.e., dispatching prefetches at the most optimal moment). Finally, the move to multi-core systems has emphasised the need for good prefetch throttling algorithms that can arbitrate and restrict the traffic coming from a growing number of prefetching units.

1.2 Main Contributions

1.2.1 Improving Prefetching Timeliness With Stream Chaining

Virtually all modern prefetching algorithms use past cache miss history information to base their predictions. However, in its original form, the global stream of past miss history contains interleaved misses from several sources (i.e, different streams of memory accesses interleaved by ILP mechanisms within the CPU). This interleaving usually leads to poor predictability of the global miss stream.

In order to tackle the poor predictability of the global miss stream, modern prefetchers usually resort to a process called *localization*. Localization refers to a clustering process in which misses are classified according to some property. The expectation is that the resulting sub-streams, called *localized streams*, will be more predictable than the global miss stream. When a new miss is registered in a localizing prefetcher, it is first localized (i.e., classified) and added to its corresponding localized stream. The prefetcher then performs its access prediction using only the miss information contained in the localized stream. This way, in a localizing prefetcher, random or noisy (i.e., non-predictable) accesses can be distinguished from regular, predictable memory access streams. Similarly, two predictable streams that appear interleaved in the global miss stream (therefore leading to poor or no predictability) appear now in two separate localized streams. Several criteria can be used to perform localization. Common strategies that have proven to work well across a variety of applications include the address of the missing instruction or the region in memory referred by the miss address.

Although localization is an useful mechanism that improves the predictability of the global miss stream, it also carries some negative side effects. In the process of localization, important chronological information about the misses is lost. While lo-

calization keeps the time ordering information for misses *within* a single stream, there is no time correlation between misses from different streams. As a consequence, current localizing prefetchers have no way of knowing how many misses from different streams might be interleaved between two misses that are stored consecutively in a given stream. Additionally, current prefetching mechanisms make use of aggressive prefetch degrees in order to hide as much as possible the effects of the memory wall. This, combined with the lack of inter-stream chronological information, leads to two undesirable effects: 1) decreased prefetching accuracy; and 2) decreased prefetch timeliness. Both effects can be attributed to the same root cause: the prefetcher issues too many prefetches for a single stream, too soon in advance. As a prefetcher predicts further into the same stream, the risk of "overrunning" the stream (i.e., predicting accesses past the natural end of the stream) and issuing wrong prefetches increases, therefore lowering the overall accuracy. Furthermore, even if all prefetches issued for a single stream are correct predictions, they might have been issued in an untimely manner (i.e., too soon) and might pollute the cache or be replaced by other misses or prefetch requests.

In order to overcome this problem, we introduce the concept of *Stream Chaining*, the first main contribution of this dissertation. The goal of stream chaining is to introduce another layer of correlation that exposes the order of activation of the different miss streams as they are used by a localizing prefetcher. This is done by linking miss streams in such a way that it reflects the core flow of misses of the application. In this context, a link between two streams implies a temporal correlation between the misses of both streams (e.g., if stream *A* is linked to stream *B*, this means that a miss localized in stream *A* is *usually* followed by a stream *B* miss). This way, for each miss, the prefetcher has two sources of information: intra-stream miss history and inter-stream activation information. The latter allows the prefetcher to predict the stream the next miss will belong to. This information can be used by the prefetcher to achieve a more balanced and timely dispatch of prefetches, issuing requests not only from the current miss stream but also from the miss streams that are predicted to be activated next.

The key to effective stream chaining lies in finding an heuristic that links the streams in such a way that it reflects the common path of misses (or stream activations) caused by the application, while at the same time leaving out spurious misses. In this dissertation we provide a concrete implementation of stream chaining we denote *Miss Graph* (MG) Prefetching. MG prefetching uses past stream activation information to generate a graph of the common stream activation paths for the current program phase.

Furthermore, the graphs generated by MG prefetchers are lightweight and can be stored in hardware with low storage requirements. Therefore, current localizing prefetchers can be adapted with minimal changes to use MG prefetching. We provide details of implementation of MG prefetching for two modern localizing prefetchers: PC/DC and C/DC. We name the resulting MG prefetchers PC/DC/MG and C/DC/MG. In the last part of this contribution we evaluate in detail the performance of these prefetchers against their non-chaining counterparts. We show how stream chaining with miss graphs significantly improves the performance of localizing prefetchers while keeping the complexity of the data structures involved low and well within hardware implementation constraints.

1.2.2 Prefetching in Multi-Core Systems with Resizable Prefetch Heaps

In the second part of this dissertation we explore prefetching in multi-core systems. As mentioned in the introduction, in the recent years the trend for increasing processor frequencies has been exchanged for an increased number of cores per processor. In virtually all multi-core architectures prefetching is performed on a per-core basis (i.e., each core has a dedicated prefetcher). This presents a new set of challenges for hardware prefetching, one of the most prominent ones being how to arbitrate access to the memory channel between an increasing number of independent prefetching engines.

Past research has produced a few prefetch throttling and filtering algorithms that regulate the aggressiveness of the prefetcher based on performance metrics and available memory bandwidth. In spite of this, only recently the problem of prefetching interference and arbitration has been researched within the context of multi-core systems. Furthermore, all past research on prefetcher arbitration in multi-core systems has focused on regulating prefetch aggressiveness at the *source* level, that is, directly setting the prefetch degree of each prefetch engine.

We introduce the concept of *Resizable Prefetch Heaps* (RPH), a novel way of arbitrating prefetches in multi-core systems at the Prefetch Request Queue (PRQ) level, with global knowledge of the state and metrics of all prefetching engines in the system. In multi-core systems, the PRQ is the data structure that holds all the prefetch requests issued by the prefetchers that have not yet been issued by the memory controller. Traditionally, this structure is implemented as a FIFO circular queue, where prefetch requests are extracted in the same order they were inserted.

The RPH is designed as a drop-in replacement of a traditional FIFO PRQ that, by contrast, works as a priority queue. In the RPH, prefetch requests are assigned a priority based on several metrics, both local to the issuing prefetcher and global to the multi-core system. This priority is used by the RPH to define an order of extraction of prefetch requests. As a result, the RPH is able to prioritize the issue of important prefetch requests over those not deemed as crucial or urgent. Additionally, instead of regulating the prefetch aggressiveness of each prefetch engine locally, we make the decision of whether to insert or drop each prefetch request at the RPH PRQ level.

Another feature of the RPH is its ability to change its size in response to the utilization of the memory bus. At times when the memory channel is saturated by demand requests, the RPH seamlessly shrinks in order to not flood the channel with more prefetching requests. Similarly, when the memory utilization is low, the RPH expands in order to be able to issue as many prefetch requests as possible.

The operation of the RPH is thus defined by two heuristics: how to assign priorities to prefetch requests and how to resize the RPH based on the current memory channel utilization. For assigning priorities, we partially base our heuristic in the principles behind a state-of-the-art throttling algorithm known as HPAC (Hierarchical Prefetcher Aggressiveness Control). We describe in detail how we construct a priority assignment formula backed by some of the operating principles of HPAC. We describe our RPH resizing heuristic, which is based solely on memory channel utilization and can be implemented with minimal hardware modifications. Additionally, we give implementation details and analyze the run-time complexity of the new hardware.

In order to evaluate the performance of the RPH, we use a selection of benchmarks in both multi-programmed and multi-threaded configurations. In contrast to previous research, we use a state-of-the-art, accurate prefetcher (C/DC) that reflects the current development of prefetching algorithms. We compare the performance of the RPH throttling against HPAC, a conventional modern throttling algorithm. We show that throttling prefetches with the RPH improves significantly the performance improvement achieved by throttling in multi-programmed and multi-threaded configurations.

1.3 Thesis Structure

Chapter 2 provides background on the current state-of-the-art in prefetching. In this chapter we introduce the basic concepts behind hardware prefetching and the metrics used to evaluate its performance. We then survey several prefetching methods, from the

fundamental and basic algorithms used in the past to the current state-of-the-art. We describe hardware implementation details such as the hardware data structures used to implement prefetching and the architectural organization of prefetching in single and multi-core systems. Additionally, we also provide information on the techniques used to throttle and filter useless prefetches.

In Chapter 3 the first main contribution of this thesis, *Stream Chaining*, is introduced. We start by setting the context and pointing out some deficiencies in current localizing prefetching methods. We then introduce the general concept of Stream Chaining, and a concrete implementation for it we call *Miss Graph Prefetching*. In the rest of this chapter we describe two new prefetching algorithms that use Miss Graph Prefetching to improve the timeliness and accuracy of their prefetches: PC/DC/MG and C/DC/MG. In Chapter 4 we evaluate the performance of Stream Chaining prefetching. We start by discussing the evaluation methodology and then evaluate in detail both PC/DC/MG and C/DC/MG against their non-chaining counterparts: PC/DC and C/DC.

In Chapter 5 we introduce the second main contribution of this thesis: *Resizable Prefetch Heaps* (RPH). First we motivate our study by characterizing the concept of prefetch throttling as a generalization of the well-known producer-consumer problem. We also describe in detail a state-of-the-art throttling mechanism known as HPAC, on which we will base some of the heuristics of the RPH. We end this chapter introducing the RPH and giving detailed information about its operation and implementation details. The performance of the RPH is evaluated in Chapter 6. As with Stream Chaining, we start by describing our evaluation methodology. We then evaluate in detail the performance of RPH compared to HPAC. We introduce the concept of *Prefetch Fairness*, a metric that allows us to evaluate the variance in prefetching performance introduced by prefetch throttling algorithms. Lastly, we evaluate the prefetch fairness of the RPH and HPAC prefetch throttling methods.

We end this dissertation with a conclusion in Chapter 7. In it, we summarize the main findings and results presented in this thesis, as well as point out future lines of work and possible further research associated with the topic. After it, we provide an Appendix with a description of all the benchmarks used in the evaluation of this work and the Bibliography referenced throughout the main text.

Chapter 2

Background

2.1 Memory Hierarchies

All but the simplest computer architectures have their memory systems composed of several layers, forming what is called a *memory hierarchy*. Each layer in the hierarchy has different characteristics regarding speed of access, density, capacity and power consumption. Generally speaking, the faster a memory technology is, the more costly it is as well. Therefore, the capacity and speed specifications of any memory technology are usually inversely correlated.

Most programs tend to access only a small portion of their address space at any given time, and they usually tend to access repeatedly the same set of memory locations. This property, called locality of reference, is crucial for understanding the usefulness of memory hierarchies. The aim of a memory hierarchy is to place the most frequently used data objects as close as possible to the element that is going to consume them, the processor. For this, fast (but costly and small) memory layers are put very close to the processor. As we move away from the processing core, we find incrementally bigger (and slower) memory layers, until finally reaching the hard disk.

Another important property of memory hierarchies is that they hide most of their implementation details to the programmer. With the exception of the hard disk, which usually requires the intervention of the operating system, and the register file, which is part of the instruction set architecture, most data flow between layers of the hierarchy is handled automatically in hardware.

Figure 2.1 shows the memory hierarchy for a conventional computer architecture, annotated with typical access latencies for each level of the hierarchy. From that figure it can be seen that as we move further away from the processor, access latencies grow

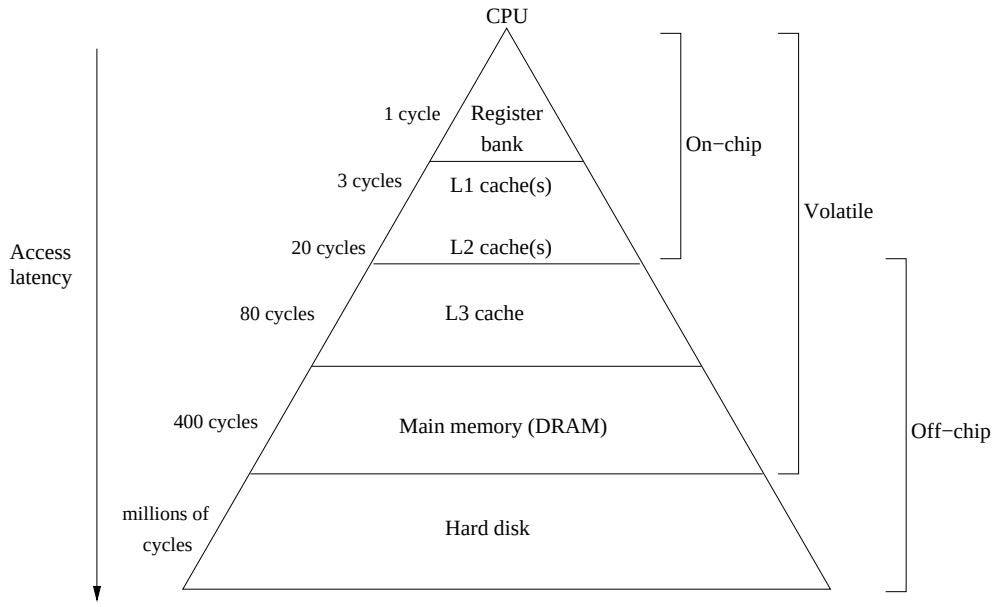


Figure 2.1: Standard memory hierarchy with typical access latencies.

larger. Conversely, memory capacities also increase as we progress further down the hierarchy. The access latency to any on-chip element is moderately small, with latencies of up to tens of cycles for the L2 cache. However, on-chip memory levels do not hold large amounts of data, with the L2 cache having capacities in the low-megabyte order.

The main memory is the first memory level which has enough capacity to store most (if not all) of the data needed by applications. The price to pay for this large capacity is a much slower access time, in the order of hundreds of cycles. This is mainly due to two factors: its off-chip placement and the use of a different memory technology (DRAM, as opposed to SRAM), needed to achieve such big capacities. This great difference in access latency is referred to as the *memory wall*, and it is explained further in Section 2.2.

The last memory layer of the hierarchy shown in Figure 2.1. This is the first non-volatile (i.e., retains data after power-off) layer of memory. Regardless of the technology used (magnetic, solid-state memory, etc.) its access latency is several orders of magnitude greater than any of the other levels in the hierarchy. Similarly, its capacity is also much greater than all the other levels combined, up to the terabyte range nowadays.

2.2 Prefetching

The term prefetching refers to the process of speculatively moving objects (usually instructions or data) up in the memory hierarchy before they get referenced by the processor. The main idea behind prefetching is to hide as much as possible the latency penalty associated by missing in one level of the memory hierarchy. While at higher levels this penalty is relatively minor (in the order of a few tens of cycles nowadays) as we move down the hierarchy it increases considerably (Figure 2.1). Of special importance is the so-called *memory wall*, which refers to the big disparity between on-chip and off-chip access latencies (i.e., to memory). Up until recently this gap in speed had been growing steadily, although in the recent years it has flatten out. However, the move to multicore processors has put new stress in off-chip communication, and with more accesses to memory being served concurrently, the memory access latency as observed by each core is expected to grow again.

Data prefetching has long been considered a successful technique to overcome the memory wall. The benefits of prefetching techniques have been recognized at least since the mid-sixties. Early studies of cache design [4] showed the advantages of fetching adjacent lines from the main memory into the cache. This is nothing but an early form of prefetching, where unreferenced lines are fetched in the hope of taking advantage of the spatial locality of the program. The IBM 370/168 (introduced in 1972) and the Amdahl 470V (introduced in 1975) were among the first architectures to implement data prefetching in hardware [5].

Prefetching can be initiated either by software or hardware. In hardware prefetching, a hardware module adjacent or included in the cache monitors the miss stream generated by the program, makes predictions about future accesses and issues prefetching requests based on those predictions. Software prefetching can be achieved by inserting *prefetch* instructions by the compiler (inline prefetching) or by running prefetch instructions in a separate thread (precomputational thread or *p-thread*) [18]. Additionally, software prefetching with helper threads can be used to perform dynamic trace analysis and insertion of prefetch instructions in the main thread [54]. This is particularly relevant in the context of execution optimization in virtual machines, where both trace analysis and instruction injection mechanisms are readily available.

Compared to software data prefetching, hardware data prefetching has two important advantages:

- It works universally and without the need of modification to the program. Soft-

ware data prefetching requires recompilation of source code.

- Hardware data prefetching has access to runtime information about the program, whereas inline software data prefetching is limited by the scope of static source code analysis. Prefetching with p-threads can have access to some runtime information about the program but with the added difficulty that the prefetching system has to always monitor the prefetching threads for divergence from the original computation path.

On the other hand, hardware data prefetchers are limited by their hardware implementation. Since they are hardware modules, they are subject to the same constraints in size, complexity and timing as all the other CPU components.

Prefetching is not exclusively applied to move data objects from memory to the last level cache. In the context of computer architecture, prefetching techniques have been used to improve the performance of instruction caches [19], I/O [20] or the TLB [9]. Moreover, data prefetching can be applied to all levels of the memory hierarchy. Current microprocessors have prefetching engines at every cache level, and frequently several specialized ones per level. However, in this dissertation we will focus on hardware prefetching operating at the last level cache (the last cache before memory). The reason for concentrating on this level is two-fold. Firstly, misses from this cache level have a higher penalty than misses from caches higher in the hierarchy, and therefore prefetching is more important for hiding the memory latency. Secondly, operating at the last level cache means that misses to it are less frequent, making timing constraints for the prefetcher less strict. This allows designing more sophisticated prefetching algorithms that realistically could not be implemented at higher cache levels.

2.3 Basic Operation of a Hardware Data Prefetcher

The prefetch module is conceptually placed between the cache it will prefetch to and the lower memory level where misses from the cache are serviced from. The prefetcher is notified of the misses generated by the cache (the *miss stream*). The miss stream is used by the prefetcher to look for predictable sequences of addresses. When one such sequence is found, the prefetcher issues a number of prefetch requests to the lower memory level. The number of prefetch requests issued is known as the *prefetch degree*, and it defines the aggressiveness of the prefetcher.

Once the prefetched data has arrived from memory, it can be either inserted directly into the cache or kept in a dedicated prefetch buffer, as proposed in [2] [7] [3]. In the latter option, any access to the cache is done in parallel with a prefetch buffer lookup. In case of a cache miss but a prefetch buffer hit, the data is moved into the cache. This has the advantage of reducing cache pollution, as useless prefetches (that are never referenced) are eventually overwritten and never move into the cache. However, for performance reasons, prefetch buffers are implemented as a fully-associative memory. Therefore their size is relatively small compared to the cache and thus there are more chances of valid prefetched data being overwritten due to lack of space in the buffer. Nowadays most microprocessors opt for inserting prefetched data directly into the cache, as it simplifies the cache and prefetcher design.

To minimize interference by the prefetcher, prefetch requests are normally given lower priority than demand misses from the cache. Additionally, the prefetched data are usually tagged with a prefetch bit to distinguish them from normal demand miss data. A prefetch bit per cache line is kept in the cache to mark lines prefetched but not yet used. When a prefetched block is accessed for the first time, a "*fake miss*" signal is sent to the prefetcher and the prefetch bit is cleared. This notifies the prefetcher that a miss would have happened had it not prefetched that block of data. This mechanism keeps the miss stream seen by the prefetcher intact and independent of the amount of data prefetched. Moreover, it allows the prefetcher to continue issuing prefetch requests even if no real L2 misses occur (i.e., as long as *fake miss* signals keep activating the prefetcher).

2.4 Prefetching Metrics

Typically three metrics are used to evaluate the performance of a prefetcher: accuracy, coverage and timeliness.

Accuracy measures the ratio of useful (i.e., eventually used by the program) prefetches to the total number of prefetches issued.

$$\text{Accuracy} = \frac{\text{Used Prefetches}}{\text{Issued prefetches}} \tag{2.1}$$

Since it is a normalized metric, accuracy will range between 0 and 1 (or 0 and 100%). When defining this metric, the term "used prefetches" has to be clarified to

indicate whether we consider a prefetch useful from the moment it is issued or from the time it reaches the cache. In the first case, we include in this metric the portion of prefetch requests that are used before they reach cache (also known as *half-misses*), and therefore failed to completely cover the memory latency. In the latter case, only prefetch requests that were in the cache at the point of being used are accounted in the metric. In this dissertation we use the latter definition of used prefetches (i.e., only those which were in the cache at the time of being used) for the accuracy metric.

Besides accuracy, another important metric to characterize a prefetcher is its coverage. The coverage of a prefetcher gives an upper bound of the fraction of misses that could be eliminated by the prefetcher operation. Coverage is defined as the ratio of issued prefetches to the total number of misses produced by the program with no prefetching.

$$\text{Coverage} = \frac{\text{Issued prefetches}}{\text{Misses without prefetching}} \tag{2.2}$$

Note that prefetching can generate new misses and generally will change the original miss stream of the application. Additionally the number of prefetches issued may be higher than the total number of misses without prefetching. For these reasons coverage, unlike accuracy, does not necessarily have to range between 0 and 1.

Timeliness is more difficult to quantify precisely. Untimely prefetch requests are those that arrive too early or too late to the cache. If a prefetch request arrives too early it might pollute the cache, as it could replace data that could be needed before it. A prefetch request issued too late will not be useful in hiding memory latency, as it will not return from memory in time to be used by the program. It could be said that a timely prefetch request is one that arrives to the cache early enough to be useful to the program but not as early as to provoke the eviction of blocks that will be referenced before it.

2.5 Hardware Prefetching Methods

2.5.1 Localization

One common issue all prefetchers have to deal with is high entropy in the global miss stream, where cache misses from several sources may be interleaved randomly, leading to poor predictability. Additionally, some of these miss sources might not be predictable at all, whereas others might. With aggressive out-of-order CPU cores being the norm nowadays, misses coming from different sources are usually interleaved and

serviced simultaneously by the cache. Consider the following code fragment for a vector sum:

```
void sumV( int *A, int *B, int *result ) {
        for (i=0; i < 10000; i++) {
                result[i] = A[i] + B[i];
        }
}
```

Normally accesses alone to vector A (or B) would generate a predictable stream of misses $A, A+n, A+2n, ...$ ($B, B+n, B+2n, ...$), for a cache block size of n. However, when both vectors are accessed the global miss stream as observed by the prefetcher is $A, B, A+n, B+n, A+2n, B+2n, ...$, which in is not as easy to predict.

In order to cope with this issue, modern prefetchers normally resort to *localization*. With localization, misses are grouped according to some property with the expectation that the resulting sub-streams are more predictable than the global miss stream. The three main classes of localization are *execution context localization*, *spatial localization* and *temporal localization* (Figure 2.2).

Execution context localization groups misses according to the instructions that generated the miss. A commonly used method in this class is grouping the misses according to the Program Counter (PC) of the missing instruction. In the above example, PC localization of the global miss stream would result in two sub-streams that are easily predictable: $A, A+n, A+2n, ...$ and $B, B+n, B+2n, ...$ PC localization gives good results and has been used in several popular prefetching algorithms, such as the Stream Prefetcher [2] [3], the Stride Prefetcher [6] and the PC/DC prefetcher [10].

Spatial localization groups misses according to the memory region they point at. Localization based on concentration zones (CZones) was proposed in [11] as an alternative to PC localization in the C/DC and AC/DC prefetchers. These prefetchers separate the stream of misses according to memory address ranges of the misses. C/DC uses memory regions of fixed size whereas AC/DC adaptively changes the size of those regions. The predictor in [13] also uses spatial localization.

Temporal localization was defined in [12] in the context of directory-based shared-memory multiprocessor systems. In this type of systems, temporal localization groups consecutive misses based on the fact that they appear in one node of the system at a time period defined by the start and end of a sequence of *coherent* read (i.e., to shared memory) misses. Although it defines a way of grouping misses based on their temporal properties, this type of localization is mostly relevant when localizing misses

from different processing sources, such as nodes in a multiprocessor system.

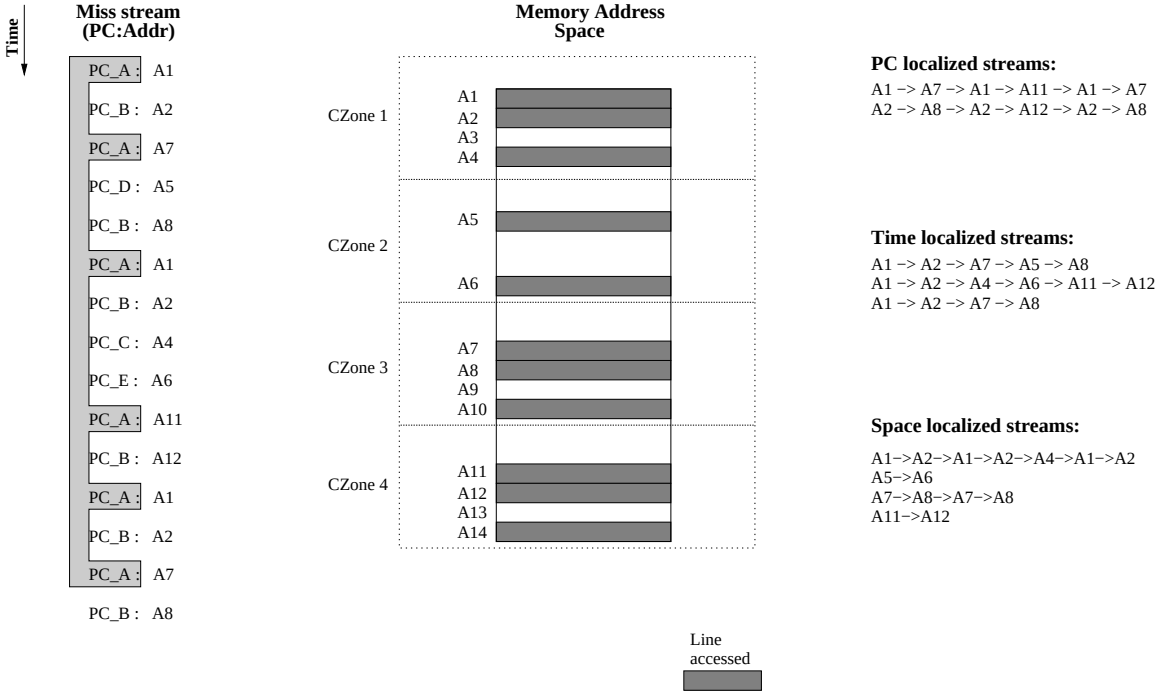


Figure 2.2: Execution context, spatial and temporal localization.

2.5.2 Correlation

Correlation is the process by which a prefetching algorithm predicts future misses based on previous historical miss information. Naturally there are several kinds of correlation heuristics. Sequential methods, described in Section 2.5.2.1, utilize minimal historical miss information and instead rely on prefetching sets of continuous memory blocks. Complex correlation methods (Section 2.5.2.2) use more detailed historical miss information that allows them to be more selective on what they prefetch. While prefetching more intelligently than sequential methods, complex correlation methods require more complex hardware implementations.

2.5.2.1 Sequential Methods

The earliest prefetch algorithms were sequential in nature, prefetching lines consecutive to the one that caused the miss. In doing so, sequential methods try to take advantage of the *spatial locality* of programs, which states that if a program references a certain memory location, it is probable that it will also reference locations nearby.

The simplest sequential prefetching algorithm requests, on a given miss, the block adjacent consecutive to the one that caused the miss [5]. This technique is sometimes called One Block Lookahead (OBL). One obvious drawback of this technique is that, in the case of a purely sequential access pattern, OBL will only prefetch every other miss (since it is necessary to have a miss in order to prefetch another cache block). This can be overcome tagging the prefetched blocks in the cache, as explained in Section 2.3.

OBL prefetching might not be enough to stop a processor from stalling, since only one block of data is prefetched at a time. More complex sequential methods usually prefetch more than one block at a time. Let us recall that the number of prefetched blocks in a single prefetcher activation is referred to as the prefetch degree (Section 2.3). One problem with prefetching with large degrees is the possibility of polluting the cache with unused prefetched data blocks. This problem is especially acute with sequential prefetching, as they do not discriminate much which blocks to prefetch. In Section 2.7 we survey methods to adaptively change the degree of prefetching and filter out cache polluting prefetches.

In [2], a variation of sequential prefetching is proposed where K blocks are brought from memory to a FIFO queue called *stream buffer*, a specialized version of a *prefetch buffer* (Section 2.3). When a demand miss references the block in the head of the queue, it is transferred to the cache and another prefetch request is sent, to be later enqueued at the tail of the queue. In [3] the usage of several stream buffers was studied as a possible replacement for second level caches. The conclusion of this analysis is that for the benchmarks evaluated, a collection of 8 streams (allocated with a LRU policy) and a prefetch degree of 2 is sufficient to predict between 50% and 90% of all the accesses to a second level cache.

The contributions in [2] and [3] form the basis of the *stream prefetcher*, a version of sequential prefetching with stream buffers that can be implemented with a simple FSM. A stream prefetcher can track several streams of memory references at the same time. Each stream is defined by an initial miss to the cache and a distance to that miss, called the *prefetch distance*. The prefetcher monitors misses to the cache that fall within the range of each stream (i.e., between the initial miss of the stream and its prefetch distance). In case of a miss being detected in that range, the prefetcher requests from memory the blocks $[A+P, A+P+K]$ ($[A-P-K, A-P]$ in a descending stream), where A is the initial miss registered for that stream, K represents the prefetch distance and P is the prefetch degree. Note that in the case of the stream prefetcher,

both the prefetch distance and the prefetch degree control the aggressiveness.

Thanks to their simple implementation and relatively good performance when dealing with simple access patterns, stream prefetchers have been implemented in several commercial microprocessors, such as in IBM's POWER4 [40] and POWER5 [41] microarchitectures. Of historical interest is Cray's decision of replacing second level caches with dedicated stream buffers and prefetchers in their T3E supercomputer [39].

2.5.2.2 Complex Correlation Methods

Complex correlation prefetching methods use past cache miss information to make predictions about future misses. Most current prefetching algorithms can be classified as belonging to this group. One of the most straightforward, known as Address Correlation ([10]) or Temporal Correlation ([12] [14]), keeps a (limited) history of past misses, chronologically sorted. When a new miss occurs, the predictor looks back in the miss history to determine if that miss has been seen before. In the affirmative case, the miss addresses recorded in the miss history as happening after this address are prefetched (Figure 2.3). Simple Address Correlation predictors only use the current miss in order to search the miss history, whereas more sophisticated methods may use a bigger context (past two or three misses for example) to improve the accuracy of the predictions at the cost of lower coverage. Address Correlation does a good job in capturing spatially irregular miss patterns that repeat in a predictable fashion. However one of its drawbacks is that it requires keeping a fairly long history of past misses in order to be effective.

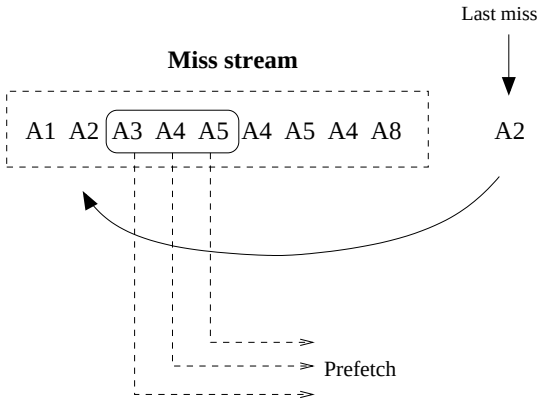


Figure 2.3: Example of Address Correlation.

Markov Prefetching generalizes Address Correlation. The concept behind Markov

Prefetching is that for a given miss address, sometimes different correlations are possible. For example, given a miss address A, the prefetcher will search the past miss history and see that in the most recent case, address B missed the cache after A. However, the miss history also indicates that there was another previous miss to address A in which the consecutive missing address was C. Markov prefetching approaches this situation modelling the miss history information as a Markov chain, hence the name. In case of successful correlation, for a given miss the Markov Prefetcher will have several records of addresses that followed it. These addresses would be prefetched starting with the most recent one (Figure 2.4).

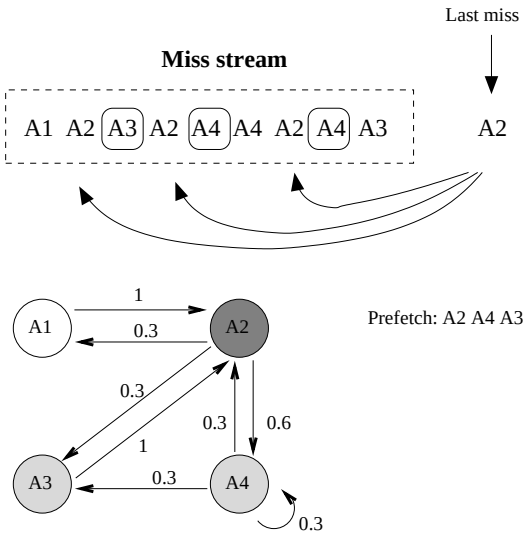


Figure 2.4: Example of Markov Prefetching.

Another way of generalizing address correlation consists of using the cache tags instead of the miss address. This idea was proposed in the *Tag Correlating Prefetcher* [46]. The main principle behind Tag Correlating Prefetching is that by using cache tags instead of addresses in the correlation, several address sequences can be generalized. The authors show that tag miss sequences (i.e., the sequence of cache tags of the cache misses) are highly repetitive, allowing the construction of a generalized predictor that requires less storage for miss information than address correlation.

Constant Stride correlation [6] aims at capturing spatially predictable miss patterns. Constant Stride predictors look at the past few misses in order to see if a pattern of constant strides is forming. If so, the prefetcher determines the stride length and starts prefetching data situated at stride multiples of the current miss address (Figure 2.5). One of the advantages of Constant Stride correlation is that it is very simple to

implement in hardware, as well as space efficient. The prefetching logic can be implemented with a very simple Finite State Machine (FSM), and the storage requirements are minimal: two registers to store the last miss address and the hypothesized stride length, and some FSM state bits per localized miss stream. On the other hand, Constant Stride predictors can only predict very simple miss address patterns.

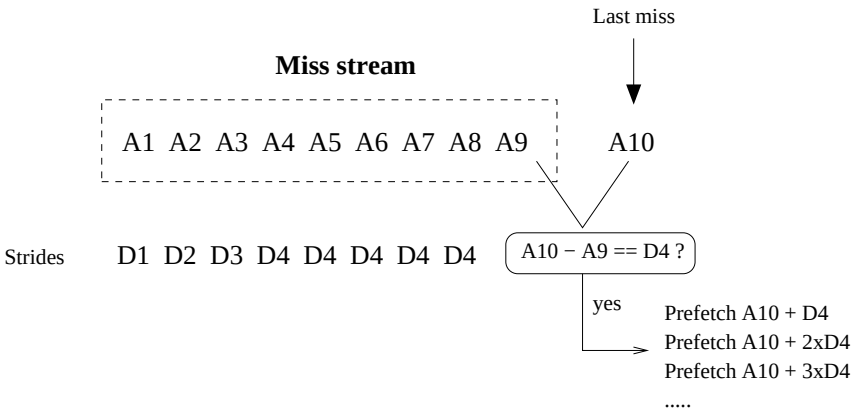


Figure 2.5: Example of Constant Stride correlation.

Delta Correlation [10] generalizes Address and Markov correlation methods. This method is based on the distance prefetching for TLB entries proposed in [9]. Like Constant Stride correlation, Delta Correlation works with the strides (deltas) between consecutive addresses in the miss stream. The Delta Correlator will try to match the last two deltas observed with the ones seen before in the delta history, much in the same way Address Correlation works. Delta Correlation is considered a generalization of Address and Markov correlations because it can predict all the accesses these two methods can, plus some other patterns. The Delta Correlator can also predict all the patterns predicted by the Constant Stride correlator, as they are just a special case of delta correlation where all the deltas are the same.

Since Delta Correlation reduces addresses to deltas, it can predict accesses to new areas of memory, as long as the access pattern is the same as one observed before. This situation can happen frequently. Normally a program will access data types or objects in a predictable manner, but these objects are scattered all around memory, depending on how or when they were allocated. Figure 2.6 shows an example of Delta Correlation that illustrates this point. Although the address $A10$ has never been encountered before by the prefetcher (within its miss history), using Delta Correlation it is possible to match its context of *address deltas* $D2, D3$ within the miss history. As

a result, the Delta Correlator can predict future references to $(A10+D4)$, $(A10+D4+D5)$ even though the strides are not homogeneous (which will confuse the Constant Stride correlator) and the miss address $A10$ is new to the prefetcher (which will make an Address Correlator fail).

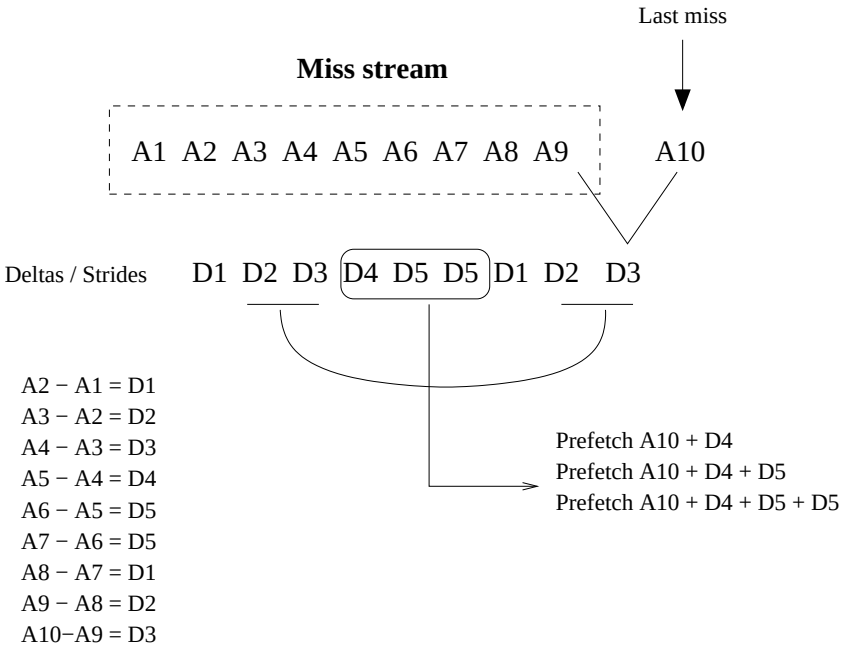


Figure 2.6: Example of Delta correlation

2.5.3 Other Approaches to Data Prefetching

Other ways to prefetch data have been proposed besides sequential and correlation based methods. In *Content Based Prefetching* [17] the prefetcher scans the data blocks brought from memory for possible addresses. Data is prefetched from those addresses in a recursive manner (i.e., newly brought prefetched data is scanned again for more addresses). The rationale behind content based prefetching is to provide an effective way to prefetch pointer-chasing irregular workloads with non-predictable spatial or temporal patterns. The content based prefetcher uses a shadow TLB which it queries in order to determine if a portion of the data block refers to a valid address.

Dead-Block Correlating Prefetchers (DBCP) [48] aim to identify which cache blocks will no longer be used and therefore are subject to eviction (“dead blocks”). Furthermore, once a dead block is detected, DBCP predicts which cache block will replace it and prefetches it. Dead blocks are predicted by tracking instruction traces

for each cache block. This is done using a fixed-size, compact trace signature based on truncating addition. These signatures form the prediction history, to which a scheme similar to address correlation is applied in order to detect when a block could be dead (this is, when the end of a previously recorded signature is detected again). Information about which block succeeds the dead block is also stored, and used for prefetching once the dead block state is reached.

Lastly, recent work in [56] tackles specifically the problem of timely dispatch of prefetch requests. In it, a new technique called Stream Timing is used to predict when the next miss of each stream will happen. A modified stride prefetcher, called Time-Aware Stride (TAS), then schedules prefetch requests to different streams depending on the predicted miss time.

2.6 Hardware Structures for Data Prefetching

In this section we describe the hardware data structures involved in hardware prefetching. In Section 2.6.1 we survey the data structures used by the prefetcher to store and retrieve past miss history information. In Section 2.6.2 we describe the Prefetch Request Queue (PRQ), the data structure that holds prefetch requests before being dispatched to memory.

2.6.1 Miss History hardware data structures

Traditionally prefetching algorithms have used tables to implement localization of the global miss stream. The table is accessed with a key such as the program counter of the miss instruction or the address of the miss. Each entry in the table contains miss information about a localized miss sub-stream. Figure 2.7 shows the table structure for the PC/DC prefetcher.

The main benefit of using tables is their simplicity in terms of implementation. They are a well understood structure that is used in other parts of the architecture such as branch prediction. On the other hand, tables are inefficient in the sense that they pre-allocate a fixed amount of history space per entry [10]. Additionally, entries in the table that are not used frequently are at risk of becoming stale and mislead the prefetcher into prefetching wrong data.

An alternative data structure, called the Global History Buffer (GHB), was proposed in [10] to overcome the deficiencies of storing miss information in tables. The

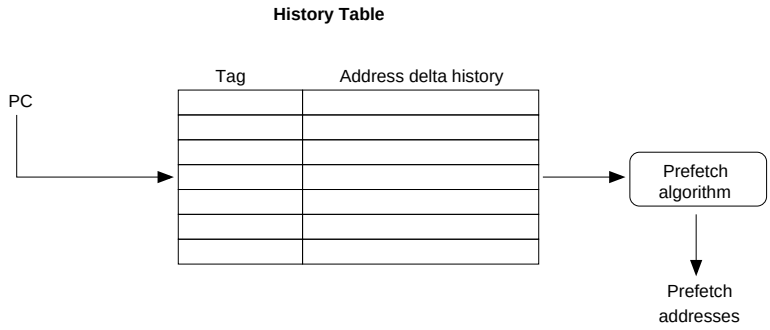


Figure 2.7: Table structure for the PC/DC prefetcher

GHB is a FIFO-like structure that stores past cache miss addresses in chronological order. It is usually implemented as a circular buffer, with a global Head Pointer pointing to the last (i.e., most recent) element on the queue.

Each entry in the GHB contains a pointer field that allows GHB entries to be connected in time-ordered linked lists forming the different localized miss streams. One possible problem due to the FIFO nature of the GHB is that an entry pointer field might point to old data that has been overwritten by newer data. This is solved by using oversized pointers, where the least significant bits are used to point to other entries of the queue and the most significant bits are used to distinguish between old and new entries [10] [11].

An Index Table (IT) is used to access the most recent address for each stream. The IT is indexed using a key appropriate for the localization scheme used (e.g., the PC of the memory access instruction that generated the miss), which allows for the implementation of different localization schemes. Figure 2.8 shows an example GHB for the PC/DC prefetcher [10].

The main advantage of the GHB lies in its FIFO queue-like behavior. Since the data structure always keeps the most recent misses and discards old entries, it solves the problem of stale entries commonly found in table-based implementations. Another advantage of the GHB is its flexibility. In [10] the authors show how the GHB can be used to implement several table-based correlation prefetchers such as the stride prefetcher, Markov prefetcher or the PC/DC delta correlation prefetcher. In [11] the authors implement two spatially-localized prefetching algorithms also using the GHB.

One drawback of using the GHB is that accessing the elements of a miss stream takes several cycles, as the hardware must navigate the linked list of GHB entries. This disallows its use in prefetchers at the highest cache levels, where the timing constraints

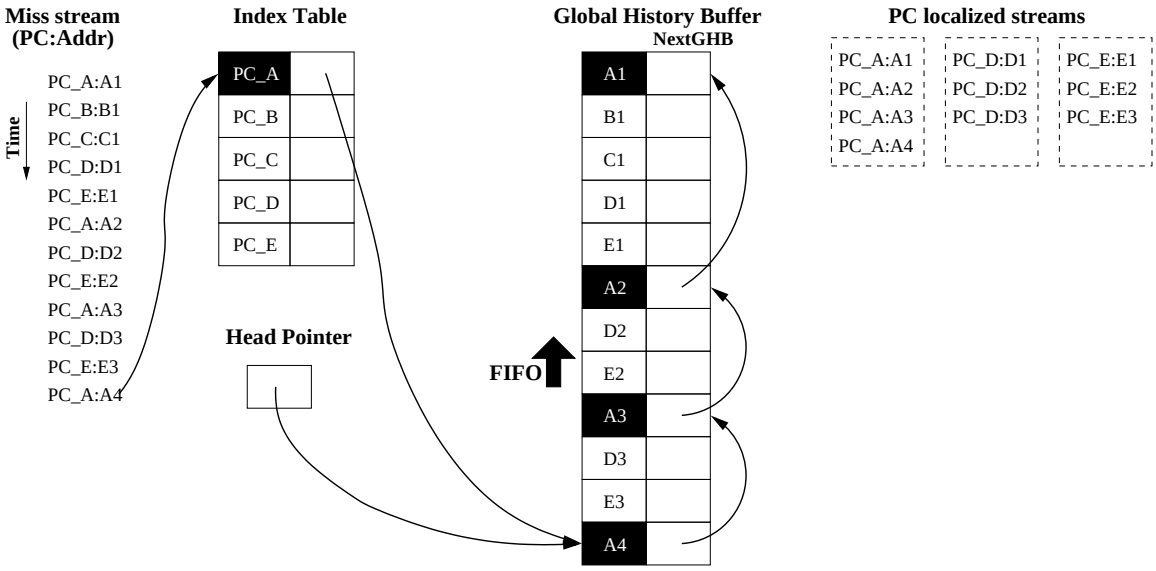


Figure 2.8: GHB for the PC/DC prefetcher.

are quite strict and the frequency of misses high. However the time between misses at lower cache levels, and specially at the last level cache, is big enough to permit a GHB hardware implementation [10] [11] [29].

2.6.2 The Prefetch Request Queue

In most architectures, prefetching requests are inserted into a hardware data structure called the prefetch queue (PRQ). The PRQ is located between the prefetcher[1] and the memory controller. This data structure holds the prefetch requests until the memory bus is free from regular memory requests, as judged by the memory controller. In this way, preference is given to demand memory requests and interference on demand requests from prefetch requests is minimized. While concrete details from commercial architectures is hard to obtain, the current consensus is that prefetch queues in current multi-cores are organized as FIFO queues and implemented as circular buffers [42, 43, 55]. This FIFO organization is also assumed by virtually all academic research in the topic of prefetching. In case of overflow, the oldest entries are dropped, and the remaining entries are (logically, but not physically) shifted to make room for the new requests.

In Chapter 5 we introduce a new PRQ organization called *Resizable Prefetch Heaps* (RPH). The RPH PRQ assigns each prefetch request a priority that reflects their relative

[1] Note that in the context of multi-core systems each core usually has its own prefetch engine.

value among the pool of pending requests. The priority of each prefetch request is then used to make decisions at the time of dispatch to the memory controller, as well as for replacement in case of overflow. This effectively turns the RPH into a logical priority queue.

2.7 Adaptive Prefetch Throttling and Filtering

In order to hide as much as possible the effects of memory latency, modern processors usually resort to aggressive hardware prefetching techniques. While beneficial for many applications, aggressive prefetching can lead to cache pollution and wasted bandwidth. This issue is even more relevant in multi-core systems, where prefetching from one core can interfere and degrade the performance of programs running on other cores. A number of approaches have been proposed to limit these side effects, usually in the form of prefetching throttlers (that adjust the prefetching aggressiveness depending on current conditions) and prefetch filters (that cancel prefetch requests that are unlikely to be beneficial). We review the current state of the art of these methods in this section, paying special attention to multi-core systems. We focus our discussion on techniques that apply for Last Level Cache (LLC) data prefetching. When we talk about multi-core systems, we refer to a general configuration where the LLC is shared among all cores, and where each core has its own prefetching engine.

2.7.1 Prefetch Throttling Techniques

Prefetch Throttling refers to the adaptive mechanisms under which the aggressiveness of a prefetcher is adjusted according to a given heuristic. This heuristic can factor in several aspects of the current system status: prefetch performance metrics, memory bandwidth consumption, prefetch pollution to the cache, interference to other cores in the case of a multi-core system, etc. The general idea behind prefetch throttling is to increase the accuracy and benefits of prefetching by adjusting its aggressiveness to its most optimal value. Naturally this increases the hardware complexity of the prefetch system, since not only we have to add the logic to implement the throttling heuristic, but also now our prefetching engine has to support dynamic reconfiguration of its aggressiveness.

Prefetch throttling mechanisms have been proposed for even the simplest prefetching methods. In [21], the authors propose a throttling mechanism for sequential prefetch-

ing in shared-memory multiprocessor systems. This mechanism tracks the number of prefetches issued as well as the fraction of those that have been used. When the number of issued prefetches reaches a certain threshold, the number of used prefetches is compared to an static threshold and a new prefetch degree is set. Therefore this method relies on prefetching accuracy to set the aggressiveness of a sequential prefetcher.

More recently, [22] proposes a probabilistic technique that tunes the aggressiveness of a stream prefetcher based on an estimated spatial locality metric. They work on a setup with a stream prefetcher residing in an on-chip memory controller. Using a data structure known as the Stream Length Histogram (SLH), they characterize the typical stream length (i.e., the length of a series of references that access consecutive blocks) for a given epoch of the running program. Then this information is used to adjust the stream prefetcher's prefetch distance and degree. Unfortunately this technique is only relevant to stream prefetchers.

A more general technique was proposed in [23]. This technique, known as Feedback Directed Prefetching (FDP), adjusts the aggressiveness of each core's prefetcher based on its accuracy and pollution side effects. The design of Feedback Directed Prefetching is general enough to be applied to different prefetching algorithms.

The Hierarchical Prefetch Aggressiveness Control (HPAC) was proposed in [24] as a generalization of FDP. Whereas FDP only takes into account metrics relevant to each prefetcher, HPAC adds another decision layer that takes into account global interactions between prefetchers. This global feedback layer tries to minimize the possible negative side effects that prefetching in one core could have for other cores. This way, HPAC is organized as a two-layer decision system, where first the global interactions between prefetchers are analyzed. If one prefetcher is found to be impacting negatively the performance of other cores (due to excessive bandwidth consumption, cache pollution) it is throttled down. In all other cases the global layer passes down control to the local control layer, which uses metrics local to the prefetcher to adapt its aggressiveness. This local layer can be any of other local-information throttling mechanisms such as FDP.

2.7.2 Prefetch Filtering Techniques

Prefetch filtering is a technique that evaluates the prefetches generated by a prefetch engine and discards those that are unlikely to be beneficial before they are sent to memory. Like Prefetch Throttling, the main aim of Prefetch Filtering is to increase the

quality of the prefetches and reduce wasted memory traffic. Unlike Prefetch Throttling, Prefetch Filtering acts independently of the prefetch engine and therefore requires little or no modifications to it.

One of the earliest prefetch filters was described in [25]. The authors describe a mechanism called Static Filter, which relies on a software profiling phase that identifies the load instructions that are likely to trigger the most successful prefetches. These loads are then put into a hardware table, which enables prefetching only for these set of instructions.

In [26], a filtering mechanism based on density vectors was proposed. Density vectors are bit vectors that track the access pattern within a region of memory at the block level. These vectors can be used to measure the predictability of spatial locality in programs. The authors show that when used in conjunction with Scheduled Region Prefetcher [8], a significant fraction of all useless prefetches can be filtered out. However, one disadvantage of this technique is that is tightly coupled with the mentioned Scheduled Region Prefetcher and is difficult to generalize for its use with other prefetching techniques.

More recently, a prefetching filter was proposed in [27] that is general enough to be used with most prefetching algorithms. This mechanism is based on history tables that hold the recent effectiveness of the prefetch requests (using 2-bit counters that track the number of times prefetch requests are referenced). For any given miss, the filtering mechanism decides whether to proceed with prefetching or not depending on the information contained in these tables. The authors propose two methods of addressing the filtering tables: by the load miss address or the PC of the instruction that generated the miss. Additionally, in order to improve the accuracy of the prediction, this lookup can be combined with context information of the Branch History Register.

2.8 Prefetching in Multi-Core Systems

In this section we introduce the organization of a generic multi-core architecture with focus on the prefetching system. We will base our study of prefetching in multi-core systems (Chapter 5) on this design. We consider an architecture with the logical organization shown in Figure 2.9(a), where a number of cores with private L1 caches share a common L2 cache. Note that the physical organization might consist of physically distributed L2 banks, or a NUCA L2, or even some other dynamic scheme for sharing private L2 caches. Moreover, any variation in L2 access times incurred by such phys-

ical organizations is secondary to our study as we focus on the large off-chip memory access penalty.

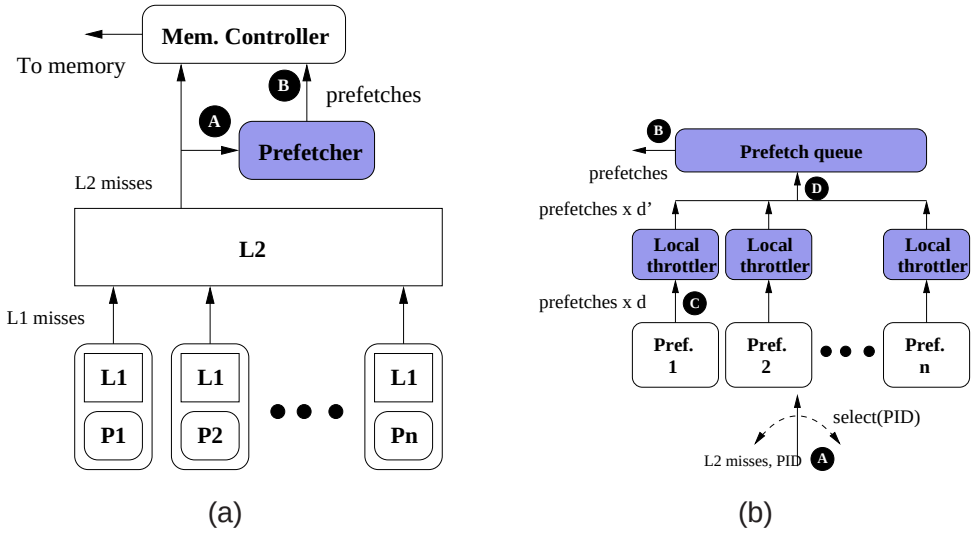


Figure 2.9: (a) Overall logical organization of a multi-core architecture; (b) Logical organization of L2 prefetchers.

In this system, a hardware prefetch engine is attached to the L2 cache and issues prefetch requests to the memory subsystem. As in most multi-cores today, the prefetch engine is logically a collection of prefetch engines, one for each processor in the system, as shown in Figure 2.9(b). In this way, each prefetch engine maintains a separate miss history for its own processor and generates prefetch requests for it, which allows each prefetcher to lock onto the individual access patterns for the application running on its processor. Each prefetch engine is associated with a prefetch throttler, that limits the rate of prefetching on each core to avoid performance degradation in case of bad prefetching behavior or scarce memory bandwidth. Again in line with current multi-cores, we assume that the prefetch engine feeds a prefetch queue, which traditionally is logically organized as a single linear (i.e., FIFO) queue, as explained in Section 2.6.2. In Chapter 5 we study in depth the design of both throttlers and the prefetch queue.

Chapter 3

Stream Chaining

3.1 Introduction

This chapter introduces Stream Chaining, the first main contribution of this dissertation. Stream Chaining provides a way to record chronological information regarding the order of activation of each miss stream in a localizing prefetcher. This information is then used to implement a more timely dispatch of prefetches and therefore increase accuracy. Overall, Stream Chaining adds a new level of design that can be used to improve current localizing prefetchers.

The rest of this chapter is organized as follows: In Section 3.2 we explain why current localizing prefetchers exhibit poor timely behavior in their prefetches. In Section 3.3 we introduce the concepts of *Stream Chaining* and an implementation for it that we call *Miss Graph Prefetching*. In Section 3.4 we introduce two new Miss Graph prefetchers based on the popular PC/DC [10] and C/DC [11] prefetching algorithms . We call these new algorithms PC/DC/MG (3.4.1) and C/DC/MG (3.4.2).

3.2 Accuracy and Timeliness in Localizing Prefetchers

As outlined in Section 2.5.1, localization is an useful strategy for organizing past miss history. Grouping misses according to a pre-set criterion helps filter out spurious accesses, which in turn translates to better correlations and higher coverage ratios.

It is important to note, however, that in the process of localizing misses important chronological information is lost. Entries from the same miss stream are ordered chronologically, but there is no time ordering between misses belonging to different miss streams.

To illustrate this problem let us assume a 100% correct ideal prefetching algorithm that uses the Program Counter (PC) for localizing misses. When this algorithm issues a number of prefetching requests $PC_i^1, PC_i^2 ... PC_i^n$ for a miss stream PC_i (i.e., the set of all misses generated by the instruction with Program Counter PC_i), it is expected that these pieces of data will be consumed in the order they have been issued. That is, PC_i^1 will be needed before PC_i^2, which in turn will be needed before PC_i^3 and so forth. On the other hand, due to the intrinsic nature of localization, there could be an indeterminate number of cache misses between any two PC_i^k and PC_i^{k+1}, as long as these requests are localized to any miss stream $PC_j \neq PC_i$. Crucially, this means that the time between any two misses for a given stream can be arbitrarily large.

As the prefetch degree increases, this problem becomes more acute. Unfortunately, prefetchers often have to resort to high prefetching degrees in order to amortize high memory latencies. In this scenario two undesirable effects can be observed:

- **Decreased accuracy:** Since many prefetches are issued for the same miss stream, the risk of some of these being incorrect increases. The prefetcher can "overrun" the actual miss stream and begin issuing incorrect prefetches. This can be because the pattern of memory accesses changes at some point or simply because the working set of the stream has been prefetched to completion.

- **Decreased timeliness:** Even if all the prefetches issued are correct and needed, they might be issued too far in advance. These data can pollute the cache, evicting other useful lines that will be needed before the prefetches and thus creating additional misses. Even if the prefetched lines do not evict useful data, since they will not be used for a long time, the chances of them being replaced by other prefetches or demand misses increases significantly.

In summary, prefetching too deep into the same miss stream might result in a waste of memory bandwidth and possible pollution to the cache. This will hinder the effectiveness of the prefetcher and can even result in application slowdown compared to setups without prefetching.

3.3 Stream Chaining

As already mentioned, the main problem with localizing prefetching schemes is that there is no chronological information relating the miss streams. This can lead to

prefetches sometimes being triggered along streams whose memory accesses appear too far apart in time, leading to untimely (premature) prefetches. Simple localization implementations, such as those based on tables (i.e., where each stream is stored as a row in a table), do not store stream activation chronology. The GHB (Section 2.6), a more advanced data structure, does contain the total timing relation among individual memory accesses of all streams. However this information is not in a format that can be readily used: the chronological miss information is stored in the GHB FIFO data structure, whereas each stream description is stored in the IT table. This means that in order to obtain the recent miss stream activation order, it is first necessary to join the information contained in both data structures. This is a costly procedure that requires at least one pass through all the GHB FIFO entries and several IT table lookups.

To overcome this problem, we introduce the concept of *Stream Chaining*. The idea behind stream chaining is to link miss streams in a way that *partially* reconstructs the chronological information in the global miss stream, such that the result corresponds directly to the *common path of misses* followed by the application. Note that in this way, what is reconstructed are sequences of *streams*, which are different from the complete sequence of *individual misses* that is found in the complete global miss stream.

Typically, localizing prefetchers have two levels of operation: a correlation heuristic to predict future misses and a localization mechanism that clusters misses in separate miss streams (Section 2.5). Stream Chaining adds an additional *third* level of operation, orthogonal to the other two, that models the chronological interactions between different localized miss streams. To accommodate this new level of operation, we extend the taxonomy introduced in [10] with a third term, so that prefetching schemes are denoted by the triple `X/Y/Z`, where `X` denotes the localization algorithm, `Y` is the correlation heuristic and the new term `Z` is the method used to link streams into groups.

Another way of thinking about stream chaining is that it attempts to predict what miss stream will be activated next in program order. In this way, a three-level prefetcher with stream chaining can predict not only the expected next misses from the current miss stream but also the expected next misses from the expected next miss streams to be activated in program order. Thus, such a prefetcher has an extra level of flexibility and can adapt to situations both in which missing memory accesses from the same miss stream are too far apart and in which missing accesses in consecutive miss streams in program order are too near. This additional level of adaptiveness can potentially improve both timeliness and accuracy with respect to traditional deep two-level prefetching.

The key to reconstructing appropriate timing information across streams is to provide a suitable stream chaining algorithm. Based on our empirical results, the flow of missing memory access instructions commonly follows stable and repeatable patterns. These patterns can be represented by a directed graph where nodes correspond to localized miss streams and edges establish a temporal order of activation between two streams, indicating that a miss in one stream is likely to be followed by a miss in the other stream. Figure 3.1 shows an example of stream chaining. In this example, we use the PC of the missing instruction to localize the global miss streams (Figure 3.1b). We then show one possible chaining of them (Figure 3.1c) that *approximates* the stream activation order found in the global miss stream (Figure 3.1a).

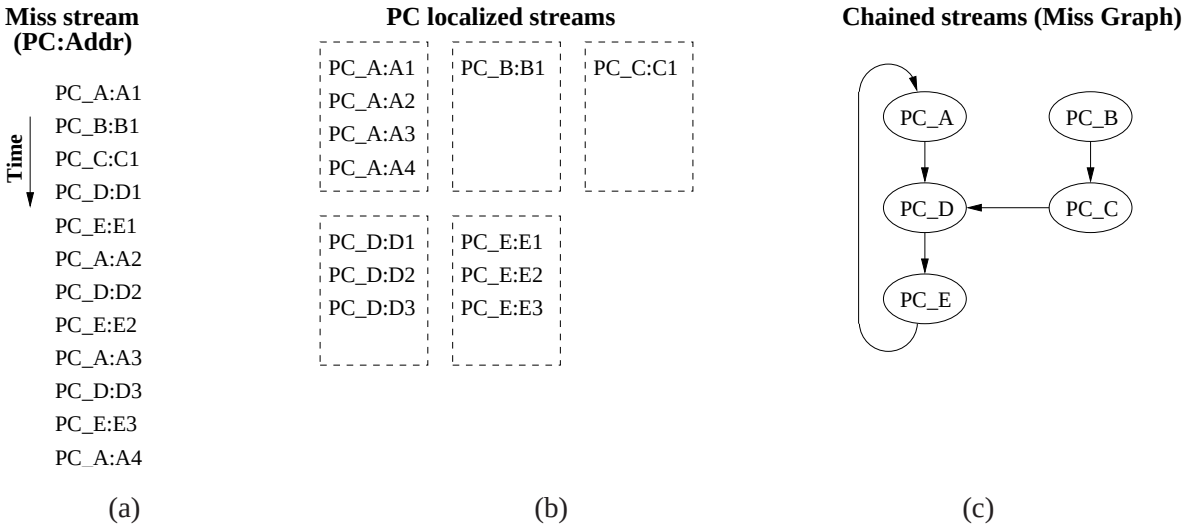


Figure 3.1: Example of Stream Chaining applied to PC localization: (a) global miss stream, (b) localized streams according to the PC of the missing instruction; and (c) one possible chaining of the localized streams.

While simple in nature, generating graphs that represent the core flow of missing memory access instructions and excludes spurious stream activations, either in infrequent control paths or that generate only occasional misses, is not trivial and is the key to a good stream chaining prefetcher. In the example from figure 3.1 linking the streams of $\mathtt{PC}_A$ and $\mathtt{PC}_B$ could be deemed inappropriate by the algorithm, as it corresponds to an infrequent flow of misses.

Therefore, the resulting graph in a stream chaining prefetcher does not contain all the possible links from the total miss sequence information in the global miss stream, but only a carefully selected set of those. Constructing the complete graph is both

impractical from the point of view of storage and processing time and yields too much information that can distract the algorithm from finding the most likely miss patterns.

The fact that the memory access chaining algorithm cannot and should not keep all links results in the graphs being disconnected or some graphs being acyclic. Note that in this way, using the information about linked streams to predict the next miss stream (and therefore, the next missing memory access) along the link may or may not be the same as predicting the next miss. However, with a good stream chaining algorithm, we have a fair amount of confidence in the fact that the stream predicted by the graph to be activated next will in fact be activated soon enough, spurious or infrequent misses not-withstanding.

In the example shown in figure 3.1, the given chaining of streams allows a prefetcher on a miss from $\mathtt{PC}_A$ to prefetch not only the next values to be consumed by $\mathtt{PC}_A$ itself but also the next value(s) to be consumed by $\mathtt{PC}_D$ (the next instruction likely to miss) or even $\mathtt{PC}_E$. Alternatively, since $\mathtt{PC}_A$ is in a cyclic graph, if the distance between its consecutive instances is too large then it could simply rely on a peer (such as $\mathtt{PC}_D$ and $\mathtt{PC}_E$) to prefetch the data it will need next.

3.4 Miss Graph Prefetching

In this section we introduce two new prefetchers that use the stream chaining approach. Both are based on popular localizing prefetchers: PC/DC [10] and C/DC [11]. Both new prefetchers use the same heuristic to chain streams. We call this heuristic Miss Graph Prefetching (MG) and therefore, using the triplet naming convention introduced in the past section, we refer to the resulting new prefetchers as PC/DC/MG and C/D-C/MG.

We use the GHB data structure to implement both the baseline prefetchers (PC/DC and C/DC) and their stream chaining counterparts (PC/DC/MG and C/DC/MG). The main reason for using the GHB is its flexibility; not only can it be used to implement different localization and correlation schemes, but implementing a stream chaining algorithm on top of it requires only trivial modifications. Moreover, in [10] and [11], it was shown that GHB-based prefetchers outperform their table-based counterparts thanks to better miss history management and elimination of stale data.

Every stream chaining prefetcher has to deal with two implementation issues: how to link streams and how to use the resulting graph to issue prefetches. For PC/DC/MG, we address these issues in Section 3.4.1 The operation of C/DC/MG is very similar,

although some aspects of the heuristic had to be modified to accommodate for the different behavior of C/DC. These changes are detailed in Section 3.4.2.

3.4.1 PC/DC/MG

PC/DC/MG is an extended version of the popular PC/DC algorithm [10]. PC/DC uses the Program Counter of the instruction that generated the miss to localize misses. Therefore each miss stream in PC/DC contains the list of misses that a given instruction (identified by its PC value) has recently generated.

3.4.1.1 Graph Construction

The key to effective stream chaining is the choice of which links to use and which to ignore. In fact, with a large enough GHB the complete chronological reconstruction of the miss stream would result in an unmanageable graph. In this section we present a simple scheme to stream chaining that results in relatively small graphs that capture the majority of miss sequences in steady-state execution. The scheme can also be easily implemented using the GHB structure.

In GHB-based prefetchers the Index Table (IT) holds pointers to every localized stream in the GHB (Section 2.6). Thus, chaining streams corresponds to linking entries in the IT. To do this, we extend the IT by adding a new pointer to each IT entry – `NextIT` – which points to the IT entry corresponding to the stream that is expected to be activated next. The `NextIT` field also includes an additional bit to signal if this pointer is valid. To identify strong (i.e., stable) links, we also add a saturating counter to each IT entry – `Ctr`. We consider a link to be stable if its associated `Ctr` counter is equal or greater than 3 (*strong link threshold*), and set the *saturation limit* of `Ctr` to 5 (both values have been determined experimentally). The operation of this counter and the role of the *strong link threshold* and *saturation limit* is explained next. Finally, we also add a new global register to the IT – `PreviousIT` – which is also a pointer to an IT entry. Figure 3.2a shows the GHB extensions in grey.

The Miss Graph algorithm builds a graph of past (temporal) correlations between localized streams as follows. Initially, `NextIT` is invalid and `Ctr` is set to zero on all entries of the IT. As misses occur, the IT and the GHB are populated as described in Section 2.6 and [10]. The new `PreviousIT` pointer is left pointing to the last stream to suffer a miss (i.e., last IT entry used). Then for a subsequent miss that activates the IT entry `IT[cur]`, we check the previous IT entry using `IT[PreviousIT]` and perform

the following operations:

```
// Check the NextIT pointer of the previous IT entry
if IT[PreviousIT]→NextIT is invalid then
    // Invalid pointer. We set it to the current entry
    IT[PreviousIT]→NextIT = IT[cur]
    IT[PreviousIT]→Ctr = 1
else
    // Valid pointer. Adjust counters depending if miss/match
    if IT[PreviousIT]→NextIT == IT[cur] then
        IT[PreviousIT]→Ctr++ (saturating increment)
    else
        IT[PreviousIT]→Ctr−−
    end if
    // If too many misspredictions, change NextIT pointer
    if IT[PreviousIT]→Ctr == 0 then
        IT[PreviousIT]→NextIT = IT[cur]
        IT[PreviousIT]→Ctr = 1
    end if
end if
PreviousIT = cur
```

By following these operations, the `NextIT` pointers in the IT form a directed graph, which can be cyclic or acyclic, can be disconnected, and in which there is only one outgoing edge from each node but possibly more than one incoming edge to a node. Figure 3.2a shows the state of the `NextIT` pointers, the `Ctr` counters, and the `PreviousIT` pointer just before the miss to $A4$ is processed. The corresponding graph is shown in Figure 3.2b. Note that, as explained in Section 3.4.1.2, we do not explicitly represent and store the graph separately; it is just shown explicitly in Figure 3.2b for clarity. Figure 3.3 shows two real examples of miss graphs from benchmarks in which we evaluate PC/DC/MG (Chapter 4). These examples were taken mid-execution and show a snapshot of the IT table graph structure, with only strong links considered (i.e., those that have a `Ctr` value greater than or equal to the *strong link threshold*, which we have empirically determined to be 3).

The graph constructed with this algorithm shows a history of correlations between localized miss streams (i.e., Program Counters of instructions generating a miss), showing which IT entry followed which in the past. The role of the saturating counters

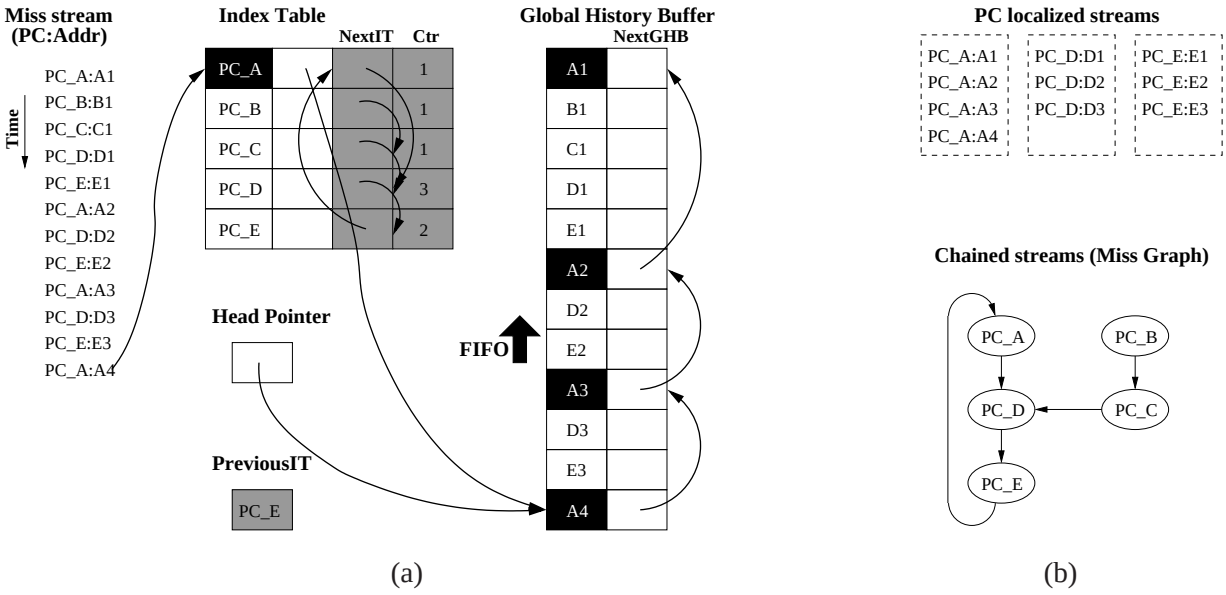
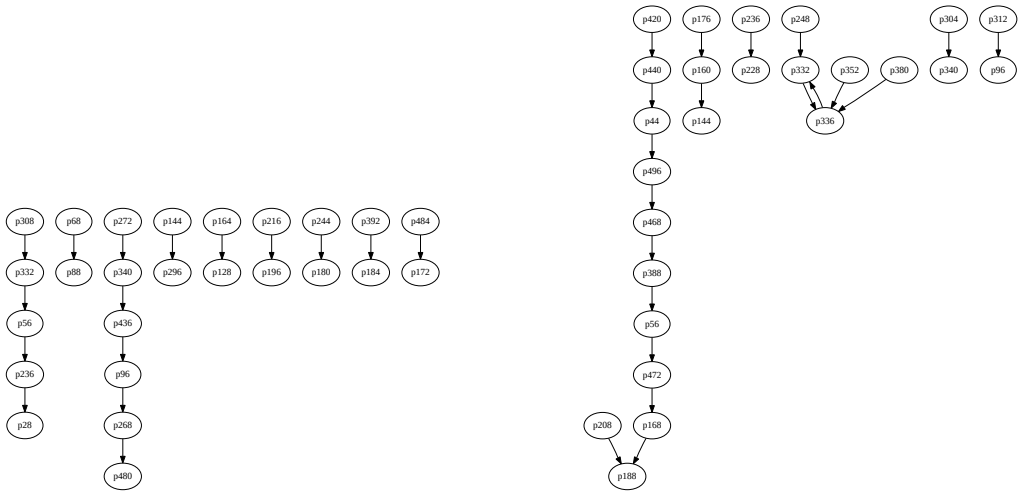


Figure 3.2: (a) PC/DC/MG extensions to the GHB. (b) Resulting Miss Graph from the example miss stream.

`Ctr` is to provide hysteresis and protection from noise from sporadic misses: by setting a minimum threshold to `Ctr` we obtain a graph with only the most stable transitions between localized streams. Next we explain in more detail how PC/DC/MG uses the extended GHB entries to prefetch along nodes in the graph.

3.4.1.2 Prefetch Operation

With the extended entries in the IT representing the miss graph, PC/DC/MG operates as follows. First, the prefetcher identifies the current miss stream, which simply involves searching the IT for an entry that matches the PC of the current missing memory access instruction. Here, unlike PC/DC, which would follow the IT pointer into the corresponding GHB stream, PC/DC/MG identifies the next stream to prefetch for by following the `NextIT` pointer in the current IT entry. So, for instance, a miss from a memory access instruction at $\texttt{PC}_A$ in Figure 3.2 will first follow the corresponding `NextIT` pointer to the stream of $\texttt{PC}_D$. For every stream that the prefetcher attempts to prefetch for, it follows the IT pointer into the corresponding GHB stream and then follows the links in the GHB entries to attempt to establish a correlation among the miss addresses. PC/DC/MG, like PC/DC, uses *delta correlation* on the addresses. If a correlation is found along the stream the prefetcher issues one prefetch along this stream. Thus, for each stream, PC/DC/MG behaves as a PC/DC with a prefetch degree of one.



(a) (b)

Figure 3.3: PC/DC/MG Miss Graph examples from real applications: (a) *bzip2* with 2MB L2. (b) *dealII* with 256KB L2 (right).

After issuing a prefetch for a given stream, the prefetcher again follows the `NextIT` pointer to the next stream to prefetch for and repeats the steps for prefetching for this stream. This process is repeated for a number of times equal to the prefetching degree parameter. In order to avoid following "weak" links into new streams, we impose a minimum threshold on the `Ctr` value below which the prefetcher will not follow the `NextIT` and will stop prefetching from further streams.

The graph construction described in Section 3.4.1.1 leads to graphs where the outgoing edge degree is no greater than one and graphs can be cyclic. Thus, starting from some node, the chains with *prefetching degree* nodes created by following the operations just described are either a linear sequence of distinct nodes or a cycle. Further, the linear sequences can either have a number of distinct nodes greater than or equal to the prefetching degree plus one, or a number of distinct nodes smaller than the degree plus one (the "plus one" comes from the fact that we skip the initial node). Figure 3.4 shows the three possible cases of graphs.

For graphs as in Figure 3.4a the operation of the prefetcher as described above is complete. For the other two cases of graphs the operation must be slightly extended. For graphs as in Figure 3.4b if we want to issue as many prefetches as the degree allows us, we would have to follow an edge back to some streams for which we have already issued a prefetch. The problem in this case is that the prefetcher would have to remember, for every revisited stream, the correlation used the last time around and

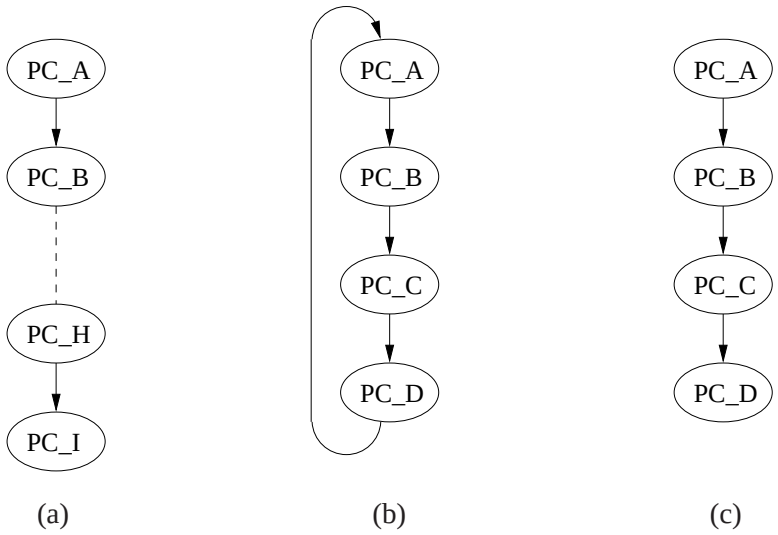


Figure 3.4: PC/DC/MG Miss Graph cases: (a) non-cyclic chain longer than the prefetch degree (8); (b) cyclic chain shorter than the prefetch degree; and (c) non-cyclic chain shorter than the prefetch degree.

the resulting address prefetched, in order to find the next delta and to compute the next address to prefetch. Instead, a simpler solution is to add a pre-pass stage where we quickly follow the "strong" `NextIT` pointers to identify whether the graph is cyclic or a long enough sequential chain. Then, if the graph is cyclic we perform the steps above except that we compute the correlations and issue more than one prefetch per stream, where each stream gets an equal share of the prefetch degree (or nearly equal when the number of nodes in the chain does not divide the prefetch degree). For graphs as in Figure 3.4c we can simply convert them to the case of Figure 3.4b by pretending that there is a back edge from the end of the chain to its beginning. Finally, if the graph consists of only the entry node then instead of starting from the next stream (which is unavailable) we simply prefetch for the current stream, basically reverting to PC/DC behavior.

The following pseudocode describes a possible implementation of the prefetch operation described above, using a pre-pass stage:

```
deg = [PREFETCHING DEGREE]
cur = [INDEX OF THE STREAM THAT GENERATED THE MISS]
chainLen = 0
visited[] // Bit array to keep track of visited streams. Set to zero initially
i = cur
// Pre-pass stage to calculate the prefetch chain length
while chainLen < deg AND visited[i] == false do
    visited[i] = true
    if IT[i]→Ctr < CtrThreshold then
        break // CtrThreshold is the counter threshold for valid links between streams
    end if
    chainLen = chainLen + 1
    i = IT[i]→NextIT
end while
// Prefetching stage
if chainLen == 0 then
    Prefetch(cur, deg) // Revert back to normal PC/DC prefetching behaviour in case of
    no stream chains
end if
degPerNode = ⌈deg/chainLen⌉ // How much to prefetch per node
totalPref = 0 // How much we have prefetched so far
while totalPref < deg do
    cur = IT[cur]→NextIT // Advance one node
    Prefetch(cur, degPerNode) // Prefetch some items for this stream
    totalPref = totalPref + degPerNode
end while
```

Since the prefetches for the different streams are generated in a single prefetcher activation, one optimization that is possible in the cases of Figures 3.4b and 3.4c is to issue the prefetches to the memory sub-system such that prefetches to consecutive streams are interleaved. Thus, for instance, in the case of Figure 3.4b we can order the prefetch requests such that the first prefetch request for $\mathtt{PC}_C$ appears right after the first request for $\mathtt{PC}_B$ and the second prefetch request for $\mathtt{PC}_C$ appears after the second prefetch request for $\mathtt{PC}_B$. This ordering is likely to be a better match to the order in which the prefetched data will be needed.

One possible advantage of three-level prefetchers with stream chaining is that given

a fixed prefetching degree budget one can divide this budget in different ways between *width* – i.e., the number of streams prefetched for – and *depth* – i.e., the number of prefetches along each stream. For instance, if deeper prefetching gives diminishing returns due to too early prefetches or decreasing accuracy, then the prefetcher can issue fewer prefetches from more streams. Alternatively, if the links between streams are too weak then the prefetcher can issue more prefetches from fewer streams. Also, we only classify links as "strong" or "weak", but one could consider finer classifications and adapt the depth for each stream depending on the "strength" of the links followed.

Like with other prefetching schemes, in order to avoid prefetched data modifying the natural stream of misses from the program a 1-bit prefetch tag is added to each cache line. This bit is set to one only in lines that come into the cache from a prefetch request and it remains set as long as the line has not yet been used. When such a line is used, a "fake" miss signal is sent to the prefetcher and the bit is reset. The prefetcher then updates its internal data structures as if it were a real miss, but no prefetch request is issued.

3.4.1.3 Hardware and Operation Complexity

As described here, PC/DC/MG uses an extension to the GHB structure. The additional storage our prefetcher requires are the `NextIT` and `Ctr` for each IT entry and a single `PreviousIT` register. As observed in [10] and in our own experience, both an IT and a GHB with 512 entries each are sufficient to capture the prefetching working set of most applications. In this case, each `NextIT` and the `PreviousIT` consist of 9 bits, as do the other pointers in the original IT and GHB, including the Head Pointer. Our experiments show that small saturation limits for `Ctr` are sufficient to capture stable links between streams and we use 3 bits.

Assuming a 32 bit PC the total hardware storage requirements of the original PC/DC algorithm are:

- IT table: 512 entries, each with a 32 bit PC field and a 9 bit GHB pointer. We can use the special PC 0 to indicate an invalid entry. $512 * (32 + 9) = 20992$ bits (2624 bytes).
- GHB FIFO queue: 512 entries, each with a 32 bit address field and a 9 bit GHB pointer. $512 * (32 + 9) = 20992$ bits (2624 bytes).
- 9 bit GHB head pointer.

- **Total:** $20992 + 20992 + 9 = 41993$ bits, 5250 bytes or approximately 5.1 KB.

For PC/DC/MG, the hardware storage requirements are:

- IT table: 512 entries, each with a 32 bit PC field, a 9 bit GHB pointer, a 9 bit `NextIT` pointer and a 3 bit saturating counter. $512 * (32 + 9 + 9 + 3) = 27136$ bits (3392 bytes).
- GHB FIFO queue: Same requirements as PC/DC. 512 entries, each with a 32 bit address field and a 9 bit GHB pointer. $512 * (32 + 9) = 20992$ bits (2624 bytes).
- 9 bit GHB head pointer and 9 bit `PrevIT` pointer.
- **Total:** $27136 + 20992 + 9 + 9 = 48146$ bits, 6019 bytes or approximately 5.9 KB.

Therefore, the additional storage requirements for PC/DC/MG compared to PC/DC are less than 1KB.

One drawback of GHB-based prefetchers is the time it takes to follow links to establish a correlation. With stream chaining, a prefetcher requires following links in multiple streams, which may further increase prefetcher operation time. The increase in operation time in comparison with the single-stream counterpart will depend on the common number of nodes in the miss graph, which in turn depends on the application. Our results suggest that the number of nodes is relatively small in practice (Section 4.4.4). In case this overhead does become a bottleneck, we note that it is possible to search for correlations in some number of streams in parallel, at the cost of replicated hardware logic. In our experiments, however, we searched for correlations from each stream sequentially.

3.4.2 C/DC/MG

C/DC/MG is the stream chaining modification of the C/DC prefetching algorithm [11]. C/DC divides the memory space into equal sized regions called CZones. The global miss stream is then localized according to the CZone in which the misses lay. A variant of C/DC called AC/DC was proposed in [11]. AC/DC adds adaptive resizing of the CZones according to the program behavior. This heuristic, while beneficial, is completely orthogonal to the modifications proposed in this dissertation, and is not considered further here.

3.4.2.1 Graph Construction

A key difference between PC and CZone localization is the number of consecutive misses localized to the same miss stream that can occur at any given time. With PC localization it is not common for two consecutive (chronologically speaking) misses to be localized in the same miss stream, since that would mean that a single instruction has generated two consecutive misses[1]. On the other hand, with CZone localization having several consecutive misses belonging to the same CZone can happen much more frequently than with PC localization. This is due to the spatial locality of the program. Said in another way, if a program is accessing a certain area of memory it is quite probable that it will access adjacent areas too, which normally will fall within the same CZone. Naturally this behavior is more common as we increase the CZone size.

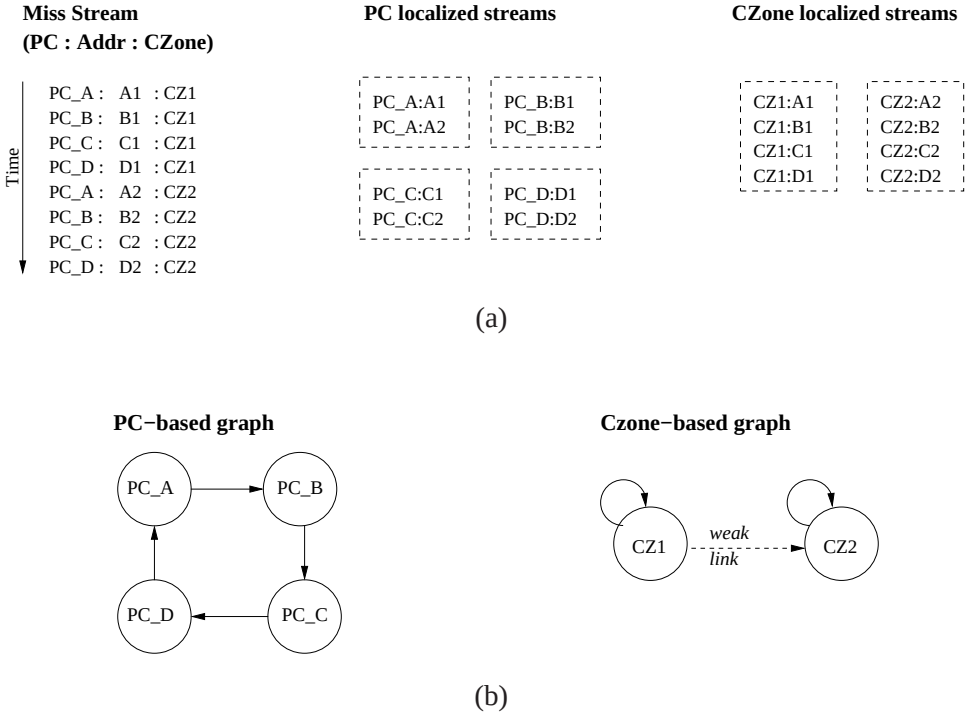


Figure 3.5: CZone transitioning problem: (a) global miss stream, PC localization and CZone localization; (b) Miss Graphs for PC and CZone localizations using the original PC/DC/MG graph construction algorithm.

[1] In the context of a RISC instruction set. With a CISC instruction set, instructions may have both operands referencing memory locations. However modern day CISC processors translate CISC instructions to RISC micro-ops before executing.

While the graph construction algorithm described in Section 3.4.1.1 could be applied to a C/DC prefetcher, the high frequency of consecutive misses localized to the same CZone would generate a graph with only trivial transitions between CZones (i.e., from each CZone to itself). This strategy would render the stream chaining variant useless, with no performance gains compared to a regular C/DC algorithm, as there would be effectively no stream chaining effect.

Figure 3.5 illustrates this problem. In 3.5a, an example miss stream is shown, as well as the result of performing both PC and CZone localization to it. In Figure 3.5b, the graph construction algorithm used for PC/DC/MG is applied for PC and CZone localization schemes. As explained in Section 3.4.1.1, this algorithm filters out infrequent transitions with the use of a saturating counter. With this algorithm, the transition between CZone 1 and CZone 2 would be considered weak, since it only happens once compared to the multiple transitions from CZone 1 to itself and from CZone 2 to itself.

One possible solution for this problem could be to just use the transitions between different CZones to construct the graph (i.e., ignore consecutive misses to the same CZone). While this strategy effectively captures the CZone transition behavior, we lose an important piece of chronological information: how many misses were assigned to each CZone. This information is vital, since after all the main difference between CZone and PC localization is the *non-trivial* number of consecutive same-CZone stream activations.

The solution we propose to tackle the CZone transition problem consists of a small modification of the graph construction algorithm for PC/DC/MG (Section 3.4.1.1). As with PC/DC/MG, we augment each GHB IT entry with a `NextIT` pointer and a `Ctr` counter, which have the same meaning as in Section 3.4.1.1. We also introduce two additional counters: the Current Repetition Counter (`CRC`) and Last Repetition Counter (`LRC`). Both counters keep track of how many repetitions (i.e., consecutive misses localized to the same CZone) happen to each CZone. Figure 3.6a shows the design of the GHB for C/DC/MG, with the proposed extensions in grey.

When a miss to a new CZone is registered by the prefetcher, the `LRC` is initialized to zero and the `CRC` is set to one. As consecutive misses localized to the same CZone occur, the `CRC` is incremented accordingly. When a CZone transition occurs (i.e., a miss from a CZone different from the one targeted by recent misses), the `CRC` is copied to the `LRC` and then reset back to zero. The transition between different CZones is treated as the transition between PCs in the PC/DC/MG algorithm.

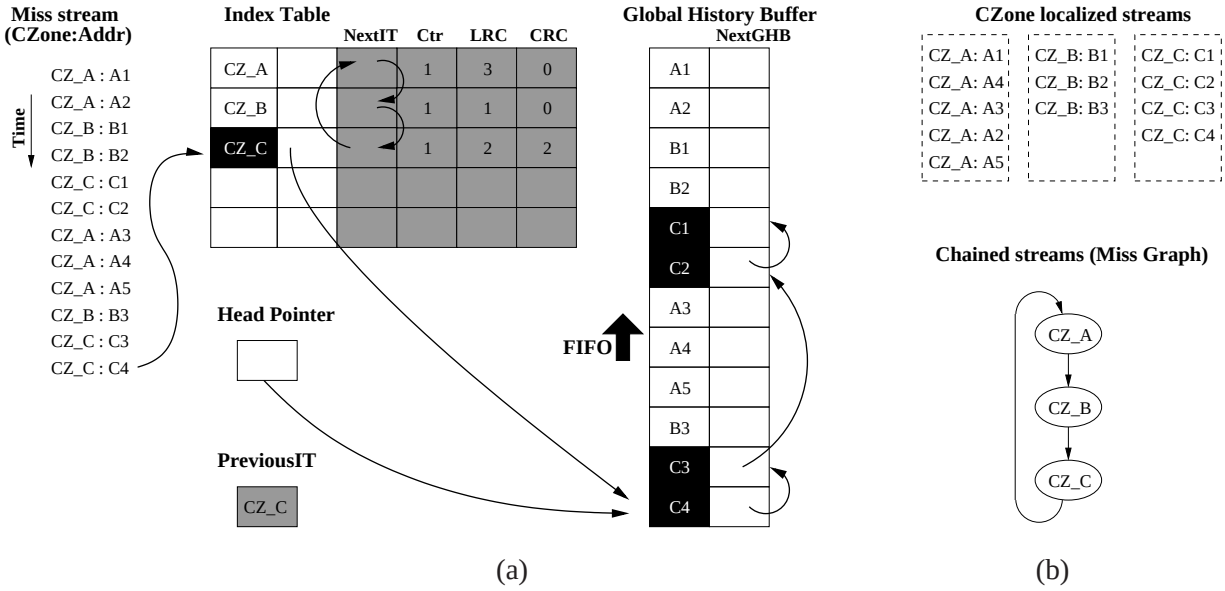


Figure 3.6: (a) C/DC/MG extensions to the GHB. (b) Resulting Miss Graph from the example miss stream.

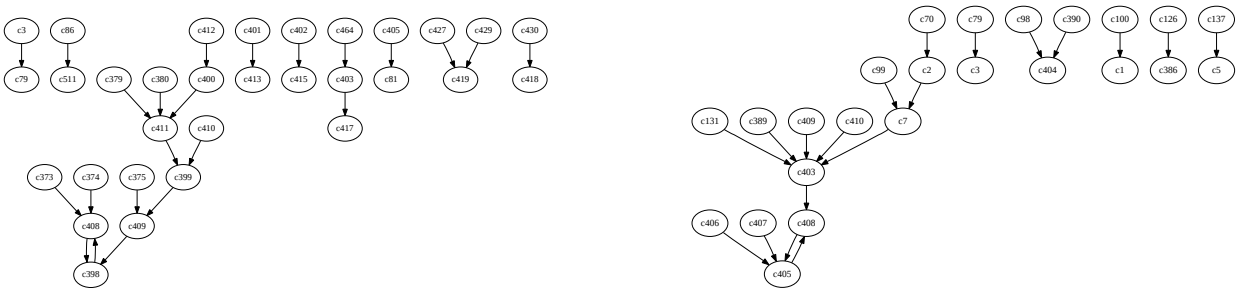
With this scheme, the graphs constructed capture both the number of repetitions on each CZone as well as the transitions between different CZones. The `LRC` counter stores how many repetitions happened to this CZone last time it was activated. The `CRC` counter is used to guide the prefetcher in how many elements to prefetch for each CZone at any given time. The details of the operation of the prefetch mechanism are discussed in Section 3.4.2.2.

As with PC/DC/MG, we provide a detailed specification of the graph construction mechanism of C/DC/MG with pseudocode:

```
// Check the NextIT pointer of the previous IT entry
if IT[PreviousIT]→NextIT is invalid then
    // Invalid pointer. We set it to the current entry
    IT[PreviousIT]→NextIT = IT[cur]
    IT[PreviousIT]→Ctr = 1
else
    // Check if we are in a CZone repetition phase
    if PreviousIT == cur then
        // Yes. Increase the CRC
        IT[cur]→CRC ++
    else
        // No, apply CZone transition logic
        // Save the CRC into the LRC
        IT[cur]→LRC = CRC
        IT[cur]→CRC = 0
        // Adjust counters depending if miss/match
        if IT[PreviousIT]→NextIT == IT[cur] then
            IT[PreviousIT]→Ctr++ (saturating increment)
        else
            IT[PreviousIT]→Ctr−−
        end if
        // If too many misspredictions, change NextIT pointer
        if IT[PreviousIT]→Ctr == 0 then
            IT[PreviousIT]→NextIT = IT[cur]
            IT[PreviousIT]→Ctr = 1
        end if
    end if
end if
PreviousIT = cur
```

3.4.2.2 Prefetch Operation

C/DC/MG issues prefetches in a similar way to PC/DC/MG (Section 3.4.1.2), but taking into account the `CRC` and `LRC` counters. The difference between the `LRC` and `CRC` indicates the number of CZone repetitions *expected* to come for this CZone. We refer to this as the *prefetch potential* of a node. The prefetch potential of each node is used

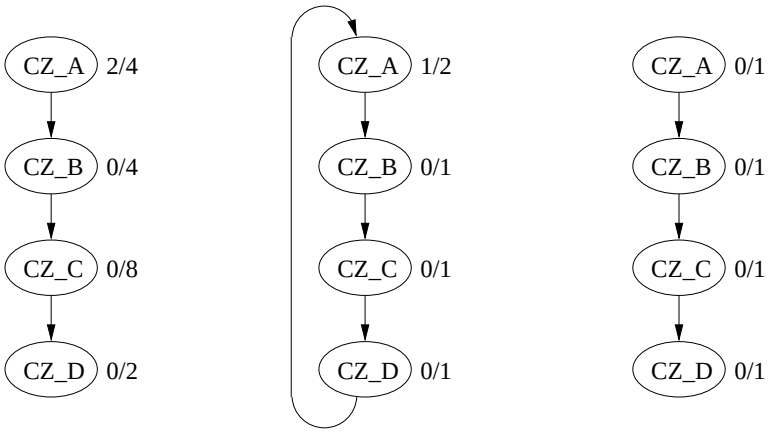


(a) (b)

Figure 3.7: C/DC/MG Miss Graph examples from real applications: (a) *bzip2* with 2MB L2; (b) *dealII* with 256KB L2.

by the prefetcher to issue several prefetches per node. The prefetching potential of a chain of nodes is simply the sum of the potentials of each node.

The graph construction algorithm proposed in Section 3.4.2.1 generates graphs similar to those generated for PC/DC/MG (Section 3.4.1.1). Therefore, there are three main operation cases C/DC/MG has to take into account (Figure 3.8).



(a) (b) (c)

Figure 3.8: C/DC/MG Miss Graph cases. The numbers adjacent to each node show the CRC/LRC counters. (a) non-cyclic chain with prefetch potential bigger than or equal to the prefetch degree (16); (b) cyclic chain with lower prefetch potential than the prefetch degree; and (c) non-cyclic chain with lower prefetch potential than the prefetch degree.

We assume a prefetch degree of 16, with the current miss localized to CZone $\texttt{CZ}_A$. The general case is shown in figure 3.8a. The prefetcher starts at the IT entry corresponding to CZone $\texttt{CZ}_A$. At that entry, $\texttt{LRC - CRC} = 4 - 2 = 2$ prefetches are issued for CZone $\texttt{CZ}_A$. The prefetcher then jumps to the next IT entry, where it repeats the same operation, issuing this time 4 prefetches for CZone $\texttt{CZ}_B$. The operation continues up

until CZone $\texttt{CZ}_D$, where the last 2 prefetches are issued. Note that, unlike PC/DC/MG, we do not skip the initial node unless the prefetch potential is equal to or less than zero (indicating that we do not expect any more misses for this CZone soon).

In 3.8b, we have a cyclic chain with less prefetching potential than the prefetch degree. In this case we act in a similar way to PC/DC/MG. We use a pre-pass stage to compute the prefetching potential of the chain. We use this number for calculating a multiplier for the prefetch potential of each node. In the example outlined in 3.8, the prefetch potential of the chain is 4. Since we attempt to issue 16 prefetches, we prefetch 4 times as many in each node. This solution is analogous to the one used for PC/DC/MG (Section 3.4.1.2). Similarly, we convert the case in Figure 3.8c to that in Figure 3.8b by adding a "virtual" back edge from $\texttt{CZ}_D$ to $\texttt{CZ}_A$.

Lastly, in the case of no graph information, C/DC/MG reverts back to standard C/DC behavior.

As with PC/DC/MG, we summarise the prefetching operation for C/DC/MG with the following pseudocode:

```
deg = [PREFETCHING DEGREE]
cur = [INDEX OF THE STREAM THAT GENERATED THE MISS]
prefPot = 0 // Prefetch potential
visited[] // Bit array to keep track of visited streams. Set to zero initially
i = cur
// Pre-pass stage to calculate the prefetch potential
while prefPot < deg AND visited[i] == false do
    visited[i] = true
    prefPot = prefPot + (IT[i]→LRC - IT[i]→CRC) // Note we do not skip the first node
    in C/DC/MG
    if IT[i]→Ctr < CtrThreshold then
        break // CtrThreshold is the counter threshold for valid links between nodes
    end if
    i = IT[i]→NextIT
end while
// Prefetching stage
if prefPot == 0 then
    Prefetch(cur, deg) // Revert back to normal C/DC prefetching behaviour in case of no
    stream chains
end if
prefMult = deg / prefPot // For cases when the prefetch potential is smaller than the
degree, prefetch more on each node
totalPref = 0 // How much we have prefetched so far
while totalPref < deg do
    cur = IT[cur]→NextIT // Advance one node
    nodePref = IT[i]→LRC - IT[i]→CRC
    Prefetch(cur, nodePref*prefMult)
    totalPref = totalPref + nodePref
end while
```

3.4.2.3 Hardware and Operation Complexity

As with PC/DC/MG, we assume that a 512 entry GHB with 512 IT entries is enough to capture the prefetching working set of most applications. Consequently we use 9 bits to implement `NextIT`, `PreviousIT` and the original GHB pointers.

From experimentation we have determined that C/DC/MG requires a smaller `Ctr` threshold counter to capture stable CZone transitions. For most cases a 1 bit `Ctr`

counter performs best. The repetition behavior of each CZone can be captured with a `CRC` and `LRC` of 3 bits each. Additionally, we have determined that a CZone size of 64KB gives the best all-round performance in our benchmarks. Therefore each CZone can be stored in 16 bits. Lastly, we have to add a valid bit for each IT entry. This was not needed in PC/DC(/MG) as the PC 0 could be used to signal an invalid entry.

The hardware storage cost for C/DC implemented using the GHB is:

- IT Table: 512 entries, each with a 16 bit CZone field, 9 bits for the GHB pointer and a valid bit. $512 * (16 + 9 + 1) = 13312$ bits (1664 bytes).
- GHB FIFO queue: 512 entries, each with a 32 bit address and a 9 bit GHB pointer. $512 * (32 + 9) = 20992$ bits (2624 bytes)
- 9 bit GHB head pointer
- **Total:** $13312 + 20992 + 9 = 34313$ bits, 4290 bytes or about 4.2 KB

For C/DC/MG, the storage requirements are:

- IT Table: 512 entries, each with a 16 bit CZone field, 9 bits for the GHB pointer, 9 bits for the `NextIT` pointer, 1 bit for the `Ctr` counter, 3 bits for the `CRC` counter, 3 bits for the `LRC` counter and a valid bit. $512 * (16 + 9 + 1 + 9 + 3 + 3 + 1) = 21504$ bits (2688 bytes)
- GHB FIFO queue: 512 entries, each with a 32 bit address and a 9 bit GHB pointer. $512 * (32 + 9) = 20992$ bits (2624 bytes)
- 9 bit GHB head pointer and 9 bit `PreviousIT` pointer
- **Total**: $21504 + 20992 + 9 + 9 = 42514$ bits, 5314 bytes or 5.2 KB

Therefore the storage requirements of C/DC/MG are 1 KB more than C/DC.

Similar to PC/DC/MG, C/DC/MG typically correlates on several streams per prefetcher activation. This increases the average operation time of the prefetcher compared to its non-chaining counterpart. However, in the case of C/DC/MG, the time complexity of the prefetch operation is not only determined by the size of the stream chain but also by how many prefetches to issue per stream: the more prefetches C/DC/MG issues per stream, the less transitions between streams it will perform (for

a given prefetch degree). This translates to a lower time overhead, since it is faster to issue additional prefetches for the same miss stream than to start a new correlation on a new miss stream.

Our results show that although on average the graphs created by C/DC/MG are larger than the ones created by PC/DC/MG, the average number of nodes in each chain (i.e., a connected component within the graph) is, as with PC/DC/MG, relatively small (Section 4.5.5). This, along with the fact that C/DC/MG transitions less often between streams, contributes to an overall lower operational complexity than PC/DC/MG. Lastly, as with PC/DC/MG, this operational overhead can be lowered even more at the cost of replicated hardware logic.

3.4.3 Table-based Alternative Implementations of PC/DC/MG and C/DC/MG

In this section we discuss an alternative implementation for PC/DC/MG and C/DC/MG that uses lookup tables instead of the GHB as the underlying hardware data structure. As described in [10], all the information needed to implement PC/DC or C/DC can be stored in a lookup table where the rows are indexed by the PC of the operation that generated the miss. As discussed in [10] and earlier in this chapter, table-based implementations have two important disadvantages: 1) they are more prone to contain stale miss history; and 2) they require more storage space and use it less efficiently. On the other hand they require a simpler hardware implementation.

3.4.3.1 Table-based Localizing Prefetchers

Figure 3.10 shows the design for a table-based implementation of the PC/DC prefetcher. The localization mechanism only defines how to index the prefetcher table, and therefore this design is also applicable to the G/DC prefetcher.

The history table of the prefetcher contains a fixed number of rows, which are addressed depending on the localization mechanism. In this way, the history table is similar to the index table of the GHB data structure. Differing from the GHB design, the miss history is stored directly in each table row, which can hold a predefined number of addresses. When a new miss address is to be inserted in a full row, the oldest address is discarded.

The storage requirements of a table-based prefetcher are mainly determined by two parameters: the number of rows in the history table and the number of miss addresses

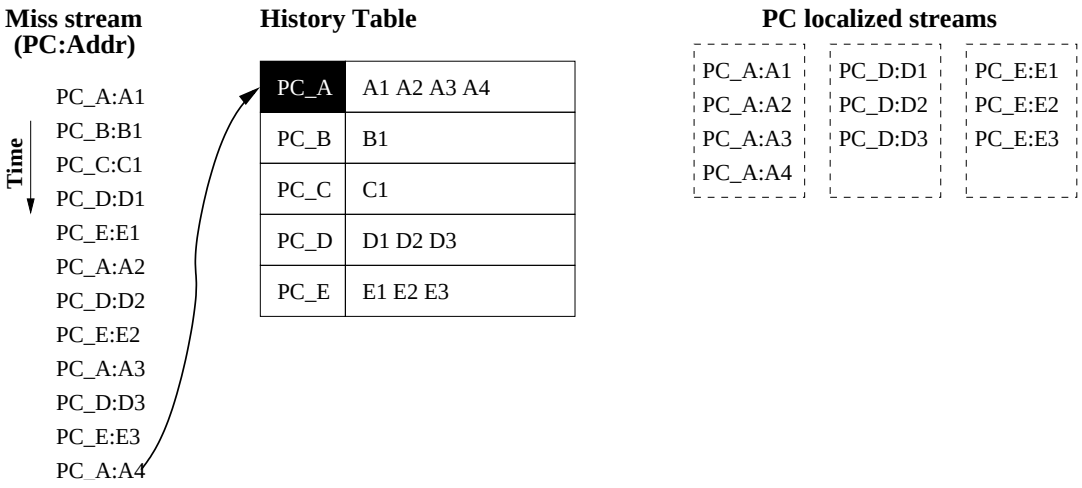


Figure 3.9: Design of a table-based PC/DC prefetcher.

stored per row. For the sake of comparison with the GHB based implementation, we set the number of rows in the table to 512, and use the parameter M_r to denote the number of miss addresses stored per row. The value M_r determines how much miss history to store per stream. This should be big enough to hold enough miss history to perform correlation on the most active (in terms of misses received) streams, even though this means storage space is wasted on streams that are not as active. For this analysis, we will use $M_r = 64$.

3.4.3.2 Table-based PC/DC/MG

Figure 3.10 shows the extensions needed to implement PC/DC/MG on table-based prefetcher hardware. As with the GHB data structure (Section 3.4.1.1), we augment the lookup table with the `NextIT` and `Ctr` fields, and add a new global register `PreviousIT`. The working mechanism and logic behind these new registers is the same as with the GHB based PC/DC/MG (Section 3.4.1.2), with `NextIT` and `PreviousIT` referencing rows in the history table.

The storage requirements for a table-based PC/DC/MG implementation, with 64 miss address entries per row, are as follows:

- History Table: 512 entries, 32 bit PC field, 64 32 bit addresses, 9 bits for the `NextIT` field and a 3 bits `Ctr` saturating counter. $512 * (32 + 64 * 32 + 9 + 3) =$ 1071104 bits (133888 bytes)
- 9 bit `PreviousIT` pointer.
- **Total:** $1071104 + 9 = 1071113$ bits, 133890 bytes or about 130.8KB

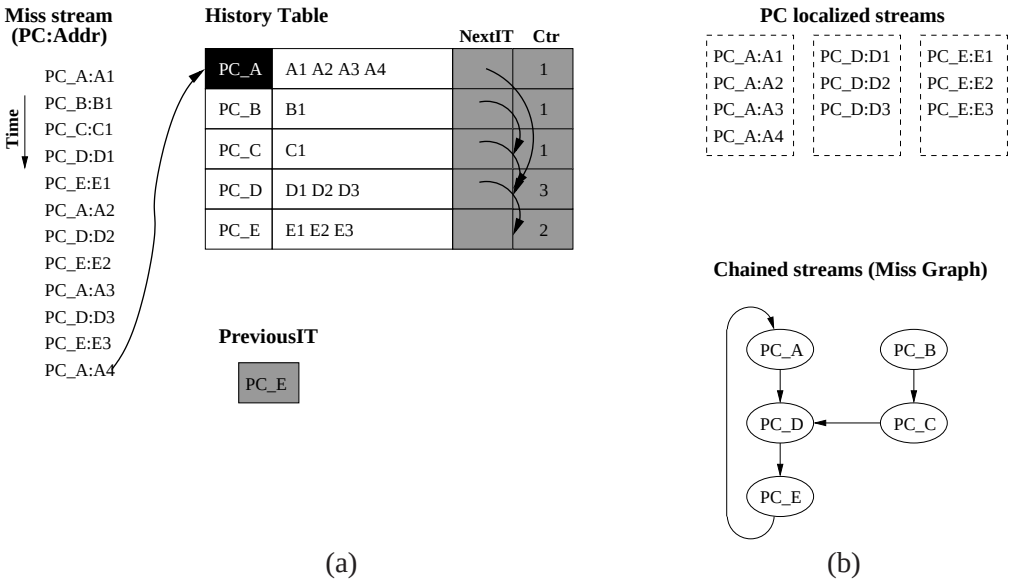


Figure 3.10: (a) PC/DC/MG extensions to the history table. (b) Resulting Miss Graph from the example miss stream.

By contrast, a table-based PC/DC implementation with the same specifications would require $512 * (32 + 64 * 32) = 1064960$ bits (133120 bytes, 130.0KB). The storage overhead of a table-based PC/DC/MG is 770 bytes.

3.4.3.3 Table-based C/DC/MG

Figure 3.11 shows the hardware extensions extensions needed to implement C/DC/MG on table-based prefetcher hardware. As with the table-version of PC/DC/MG, C/DC/MG on table-based prefetchers uses the same hardware additions than its GHB-based version (Section 3.4.2.1), and the same working logic (Section 3.4.2.2).

The storage requirements for a table-based C/DC/MG implementation, with 64 miss address entries per row, are as follows:

- History Table: 512 entries, each with a 16bit CZone field, 64 32 bit miss addresses, 9 bit `NextIT` pointer, 3 bits for each of the `CRC` and `LRC` counters, 1 bit for the `Ctr` counter and 1 bit to indicate if the entry is in use. $512 * (16 + 32 * 64 + 9 + 3 + 3 + 1 + 1) = 1065472$ bits (133184 bytes).
- 9 bit `PreviousIT` pointer.
- **Total:** $1065472 + 9 = 1065481$ bits, 133186 bytes or about 130KB

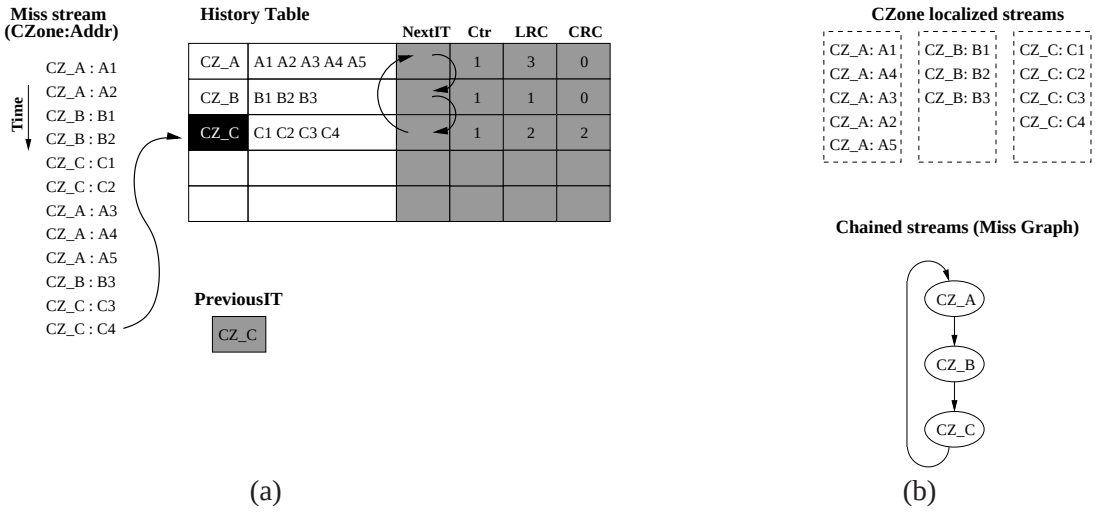


Figure 3.11: (a) C/DC/MG extensions to the history table. (b) Resulting Miss Graph from the example miss stream.

A table-based C/DC implementation would require $512 * (16 + 64 * 32 + 1) = 1057280$ bits (132160 bytes, about 129KB). Therefore the storage overhead of C/D-C/MG is 1026 bytes, just over one kilobyte.

Chapter 4

Evaluation of Stream Chaining Prefetchers

4.1 Simulation Setup

We use SESC [30] for all our simulations. SESC is a fast cycle-accurate architectural simulator that can be extended easily thanks to a modular design implemented in C++. SESC does not provide full system simulation; programs have to be recompiled with a customized cross-compiler that constructs tailor-made binaries suitable for simulation. System calls are trapped and executed in the host system, with the results sent back to the simulation environment. Cross-compiler tools provided in the SESC environment allow compilation of C, C++ and Fortran 77 programs.

For evaluating stream chaining prefetchers, we simulate a 4-issue out-of-order superscalar processor with separate L1 instruction and data caches, and a unified L2 cache on chip. All simulated caches are non-blocking. Table 4.1 lists the architectural parameters of the simulated system. We simulate mainly two L2 cache sizes: 256KB and 2MB. The former is representative of the cache share expected in a fully loaded multi-core setup. The latter reflects the case when only a single processor is active.

We fast forward each simulation by one billion instructions and then we simulate in detail and collect statistics for the next one billion instructions. Special care was taken to confirm that the first billion instruction fast-track is enough to skip the data loading phases of the benchmarks, thus making sure that the detailed simulation starts in a relevant computation phase.

Parameter	Value
Core Frequency	5GHz
Fetch/Issue/Retire Width	6/ 4/ 4
I-Window/ROB	80/ 152
Branch Predictor	64Kbit 2BcgSkew
BTB/RAS	2K entries, 2-way/ 32 entries
Minimum misprediction	20 cycles
Ld/St queue	108
L1 ICache	64KB, 2-way, 64B lines, 2 cycles
L1 DCache	64KB, 4-way, 64B lines, 2 cycles
L1 MSHR's	4
L1-L2 bus	64bits
L2 Cache	256KB/2048KB, 8-way, 64B lines, 13/18 cycles
L2-Memory Bus	64bits, 1.25Ghz
Main Memory	400 cycles
Prefetch degree	1/4/8/16
IT	512 entries, 1 cycle
GHB	512 entries, 5+1*hop cycles

Table 4.1: Simulated architectural parameters.

4.2 Benchmarks

For evaluating the prefetchers we use benchmarks from SPEC CPU2006 [31] and the BioBench benchmark suite [32]. Due to limitations with the simulator tools, not all of the benchmarks can be cross-compiled for use with SESC. A detailed description of the benchmarks used in this evaluation is provided in Appendix A.

All benchmarks were compiled with the GNU compiler GCC v3.4 using the O3 optimization level. They were run using the supplied reference input data set with the exception of *clustalw* from BioBench, for which the larger input data from the benchmark *hmmer* (also from BioBench) was used.

4.3 Benchmark Characterization

4.3.1 L2 Cache Size Sensitivity

Figure 4.1 shows the performance without prefetching for various cache sizes. Besides 256KB and 2048KB, we include results for a 512KB L2 to gain further insight into the working set size of each benchmark. The performance results are given as IPC (Instructions per Cycle) normalized to the IPC obtained with an ideal L2 cache (i.e., a cache with 100% hit rate). Figure 4.2 shows the L2 cache Read Hit Rate (RHR) for the same set of cache sizes. Lastly, we show in Figure 4.3 the number of accesses (hits or

misses) the L2 cache receives. Since this metric is fairly independent of the L2 cache size (results for all cache sizes evaluated here differ 1% or less), we show results using a perfect L2 cache.

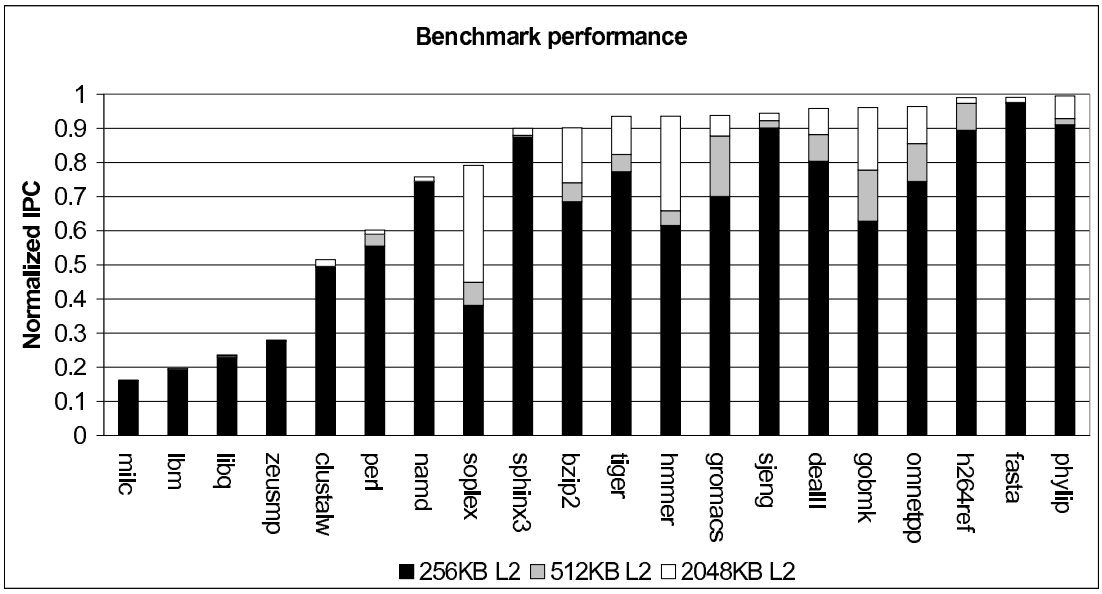


Figure 4.1: L2 cache size sensitivity for 256KB, 512KB and 2MB L2 caches.

For a cache size of 256KB, 3 out of the 20 benchmarks (*sjeng, fasta, phylip*) are within 10% of the performance of an ideal L2. These benchmarks are the less likely to benefit from prefetching. Both *sjeng* and *fasta* show a high RHR (75.4% and 84.3%), which explains their performance close to an ideal L2. On the other hand, the high performance but relatively low RHR of *phylip* (55.3%) is explained by its very low L2 usage: 31 accesses per 1K instructions, as shown in Figure 4.3. Doubling the cache size to 512KB only adds another benchmark to this list (*h264ref*), and in general only improves the performance of two benchmarks in a considerable manner (*gromacs, gobmk*). This suggest that for the scenario that a 256KB L2 cache represents (multi-core environment with several processors competing for cache space) doubling the size of the cache helps but does not cause a radical performance improvement.

For a cache size of 2048KB, 12 benchmarks (from *sphinx3* to *phylip*, in the order shown in Figure 4.1) achieve performance within 10% of a perfect L2 cache. However, the IPC improvement of increasing the cache from 256KB to 2048KB is below 30% except for two benchmarks: *hmmer* (45.1%) and *soplex* (89.9%). The L2 RHR stays above 80% for this group of benchmarks.

In 6 out of 20 benchmarks (*milc, lbm, libquantum, zeusmp, clustalw, perlbench*)

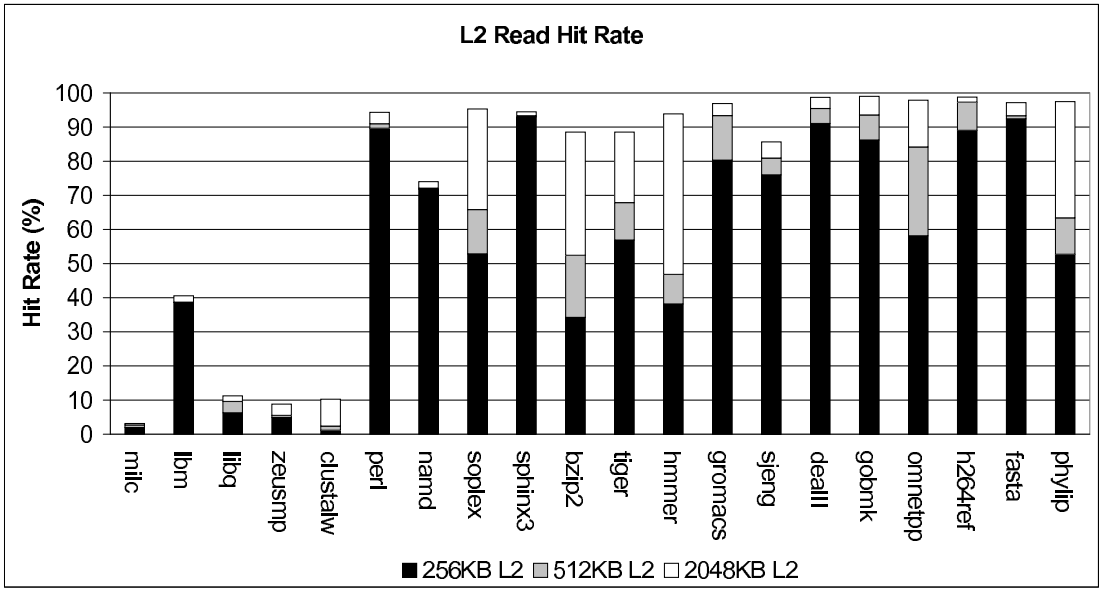


Figure 4.2: L2 cache Read Hit Rate for 256KB, 512KB and 2MB L2 caches.

we observe performance below 60% of an ideal cache, even with a large 2048KB L2 cache size. Naturally these benchmarks are the ones that are more likely to benefit from prefetching. All these benchmarks except *perlbench* show very low cache RHR. Although *perlbench* shows a high RHR, even for the smaller of the cache sizes, this benchmark shows a very high number of accesses to the L2: 191 per 1K instructions executed (Figure 4.3), more than double the average of all the benchmarks (87 L2 accesses per 1K instructions). In *milc* and *lbm* there is little improvement in the hit rate as we increase the size of the L2, indicating a memory footprint that is not easily captured by a cache memory. *libquantum*, *zeusmp* and *clustalw* show significant increases in hit rates for larger L2 sizes, but even with the largest cache size the RHR is around 10% at best. This indicates that these benchmarks operate with a very large working set.

4.3.2 Miss Distances

One characteristic that helps understand the behavior of non-chaining and chaining prefetch schemes is the distances between consecutive misses. Using an execution *without* prefetching, we measure the number of cycles between consecutive misses, coming from any instruction (*global* miss distance), from the same instruction instruction (*PC* miss distance) or targeting the same CZone (*CZone* miss distance). For the latter metric we use 64KB CZones. Figures 4.4 and 4.5 show the miss distances

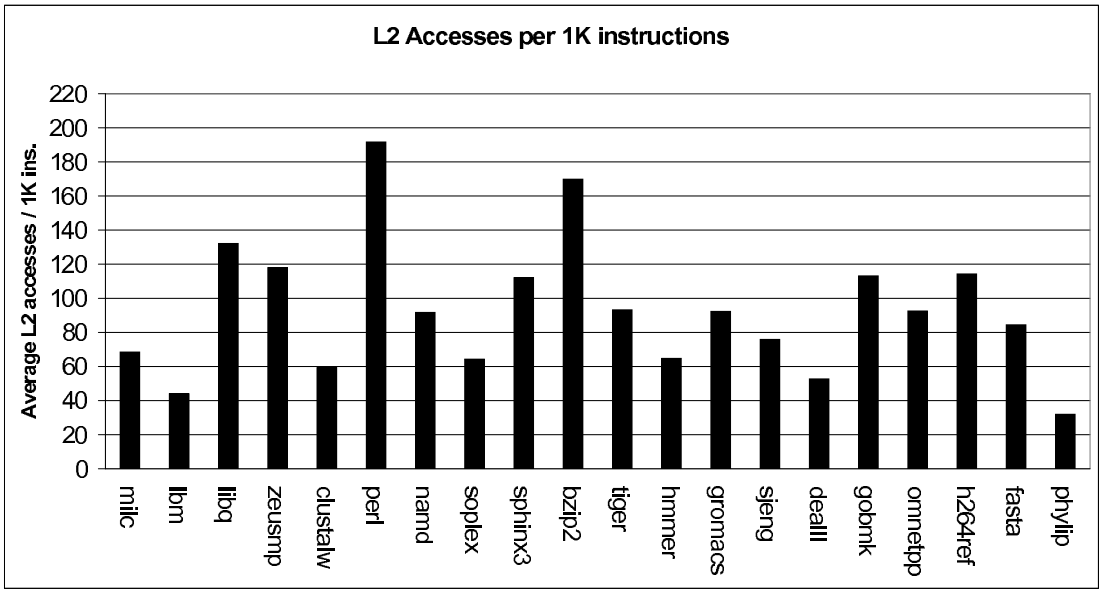


Figure 4.3: Number of L2 cache accesses per 1K instructions.

grouped by ranges of number of processor cycles for 256KB and 2MB L2 cache sizes, respectively.

From this figure we can see that often a large fraction of L2 misses occur thousands or even hundreds of thousands of cycles apart. Moreover, the fraction of such distant misses often increases with cache sizes. More importantly, the fraction of such distant misses is significantly larger when we consider PC or CZone miss distances. Again, these results suggest that PC/DC and C/DC with large degrees of prefetching, while still being able to eliminate some more misses, may issue such deep prefetches too early. Lastly, CZone average distances tend to be shorter than per-PC distances due to the skewing effect of CZone repetition (as explained in Section 3.4.2.1), but this distance can be much higher when transitions between CZones occur.

4.4 PC/DC/MG

4.4.1 Performance and Traffic

Figures 4.6 and 4.7 show the performance improvement (gray portion of each bar) of PC/DC and PC/DC/MG for 256KB and 2048KB L2 caches sizes, respectively. A prefetch degree of 16 was used. Performance is given as IPC normalized to the IPC obtained with an ideal L2 cache. Since prefetching usually involves an increase of

memory bandwidth, we also show this metric in the same plot (line plot with right Y axis). We measure the total bus traffic (reads and writes, command and data channels) and plot it as a percentage of the traffic observed with a configuration without prefetching.

From both graphs it can be seen that the majority of applications benefit from prefetching. The exception is *omnetpp*, for which prefetching actually degrades performance with a 256KB L2.

As expected, the configuration with a lower cache size benefits more from prefetching. However even with the large 2048KB L2 several applications experience large speedups.

For a cache size of 256KB, PC/DC/MG outperforms its non stream chaining variant in 14 out of 20 benchmarks. For the *sjeng* benchmark, prefetching does not improve performance regardless of the algorithm used. PC/DC and PC/DC/MG both slightly benefit *fasta*, but being so close to the ideal L2 performance there is little room for improvement. In *omnetpp*, as mentioned before, prefetching degrades performance, with the degradation being slightly worse with PC/DC/MG. Only two benchmarks, *tiger* and *hmmer*, benefit more from PC/DC than from PC/DC/MG. However, the performance degradation is small: 1.1% in *tiger* and 2.1% in *hmmer*.

With respect to traffic, both PC/DC and PC/DC/MG have similar overhead, with two exceptions. The benchmark *lbm* sees a substantial decrease in memory traffic when using PC/DC/MG. This is due to a notable reduction in L2 writeback traffic in PC/DC/MG (the read traffic stays the same in PC/DC and PC/DC/MG in this case). On the other hand, *omnetpp* shows a significant increase in memory traffic when using PC/DC/MG compared to PC/DC. With both PC/DC and PC/DC/MG, prefetching increases the memory traffic to about 200% percent of the non-prefetching configuration, without giving any performance benefits.

Similar to the 256KB L2 configuration, with a cache size of 2MB, PC/DC/MG outperforms PC/DC in 14 out of 20 benchmarks. As it is expected, a bigger cache size pushes the performance without prefetching of most applications closer to that of an ideal L2 cache. The benchmarks *h264ref*, *fasta* and *phylip* are too close to an ideal L2 performance to benefit from any prefetching. As with a 256KB L2, the benchmark *sjeng* does not benefit from either prefetching algorithm. Different to the 256KB L2 case, *omnetpp*'s performance is not degraded by prefetching. In this case, PC/DC prefetching does not improve performance but PC/DC/MG does, bringing it very close to the ideal L2 mark. Lastly, only *hmmer* benefits (very slightly) more from PC/DC

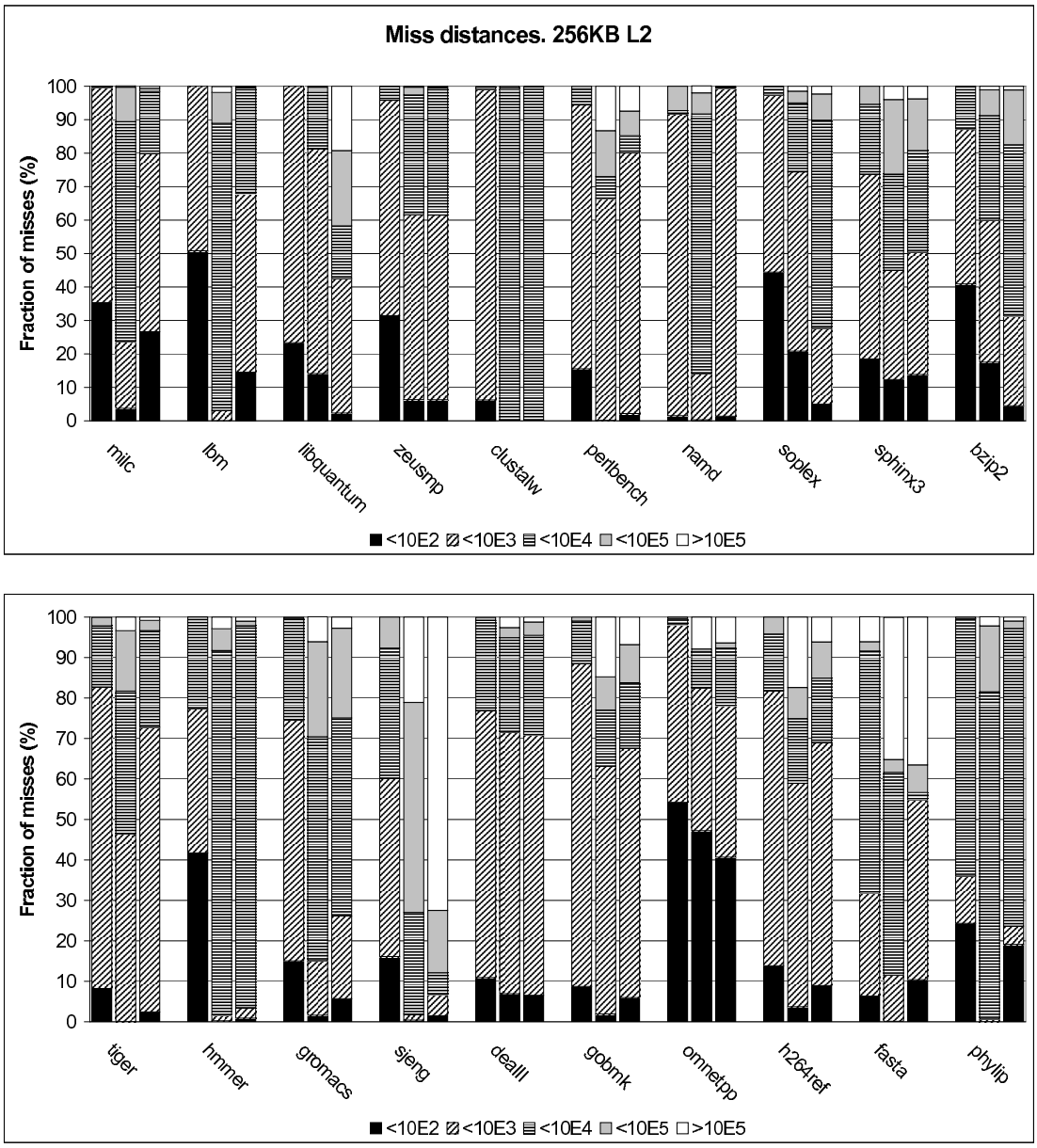


Figure 4.4: Miss distances for 256KB L2. Figure shows global (left), per PC (middle) and per CZone (right) miss distances.

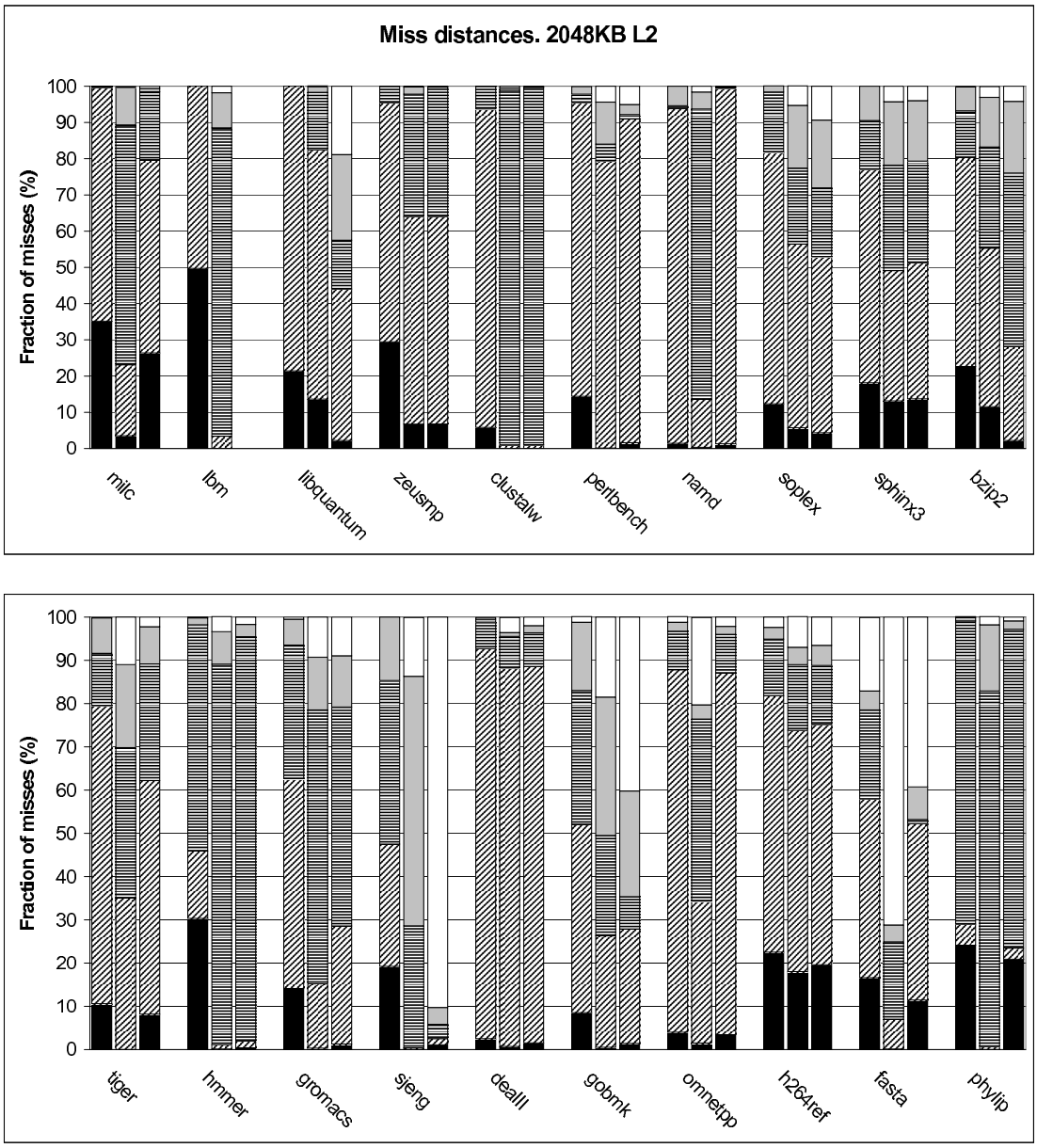


Figure 4.5: Miss distances for 2MB L2. Figure shows global (left), per PC (middle) and per CZone (right) miss distances.

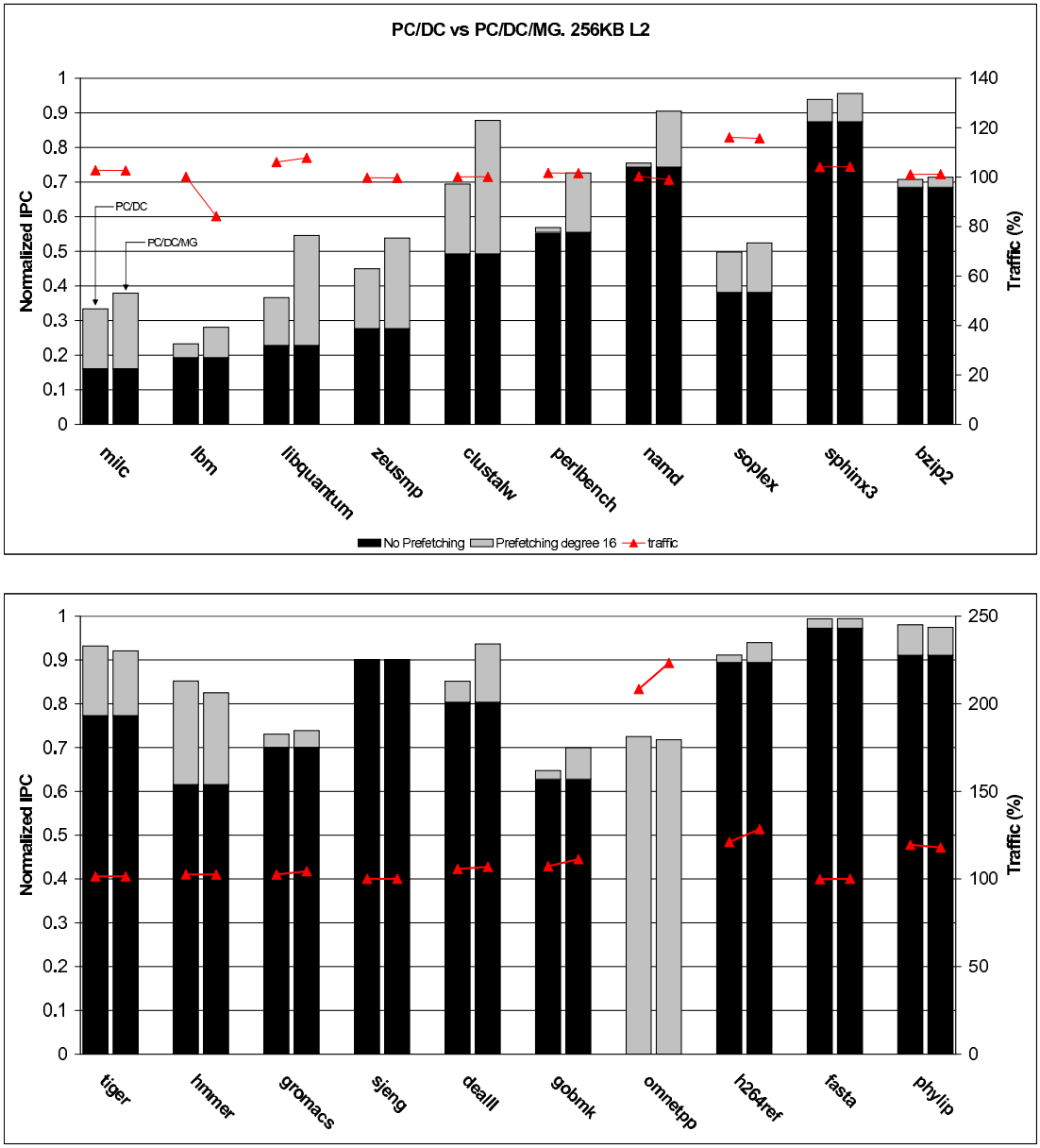


Figure 4.6: PC/DC (left bar) and PC/DC/MG (right bar) performance and traffic. 256KB L2 cache.

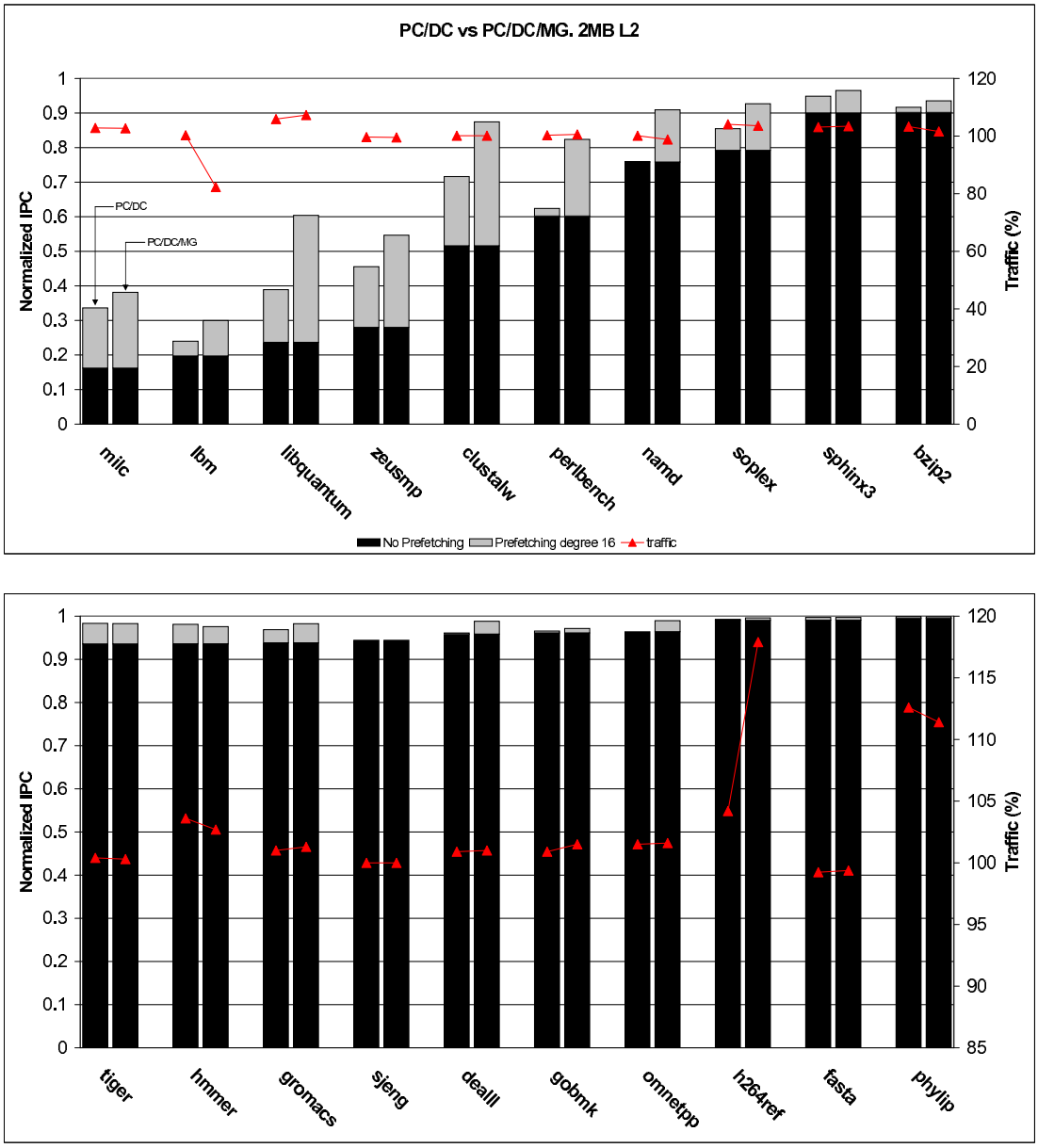


Figure 4.7: PC/DC (left bar) and PC/DC/MG (right bar) performance and traffic. 2MB L2 cache.

than from PC/DC/MG. However, the performance degradation compared to PC/DC is only 0.5%.

Not surprisingly, the traffic increase due to prefetching is significantly lower in the 2MB L2 configuration. Lower number of L2 misses (compared to a 256KB L2) translate into lesser prefetcher activations. Both PC/DC and PC/DC/MG exhibit roughly the same increase in bus traffic, with two exceptions. First, the benchmark *lbm* sees a substantial decrease in memory traffic when PC/DC/MG is used. This is due to the same causes as with the 256KB L2 cache (decreased L2 writeback traffic). Secondly, the total traffic in *h264ref* rises from 104% (of the non-prefetching traffic) for PC/DC to 117% for PC/DC/MG. However, the total memory traffic generated by this benchmark is already very low (thus it is very close to the ideal L2 performance), therefore a small number of additional prefetch requests (as is the case with PC/DC/MG) will alter the traffic increase numbers significantly.

4.4.2 Coverage and Accuracy

Figures 4.8 and 4.9 show the coverage of PC/DC and PC/DC/MG with prefetch degrees of 1, 4, 8 and 16. Naturally, the coverage varies across benchmarks, but overall PC/DC/MG often offers higher coverage for the same degree. This increased coverage is mainly due to the cross-stream nature of PC/DC/MG. When PC/DC cannot correlate a stream, no prefetches are generated and this lowers the overall coverage of the prefetcher. By contrast, PC/DC/MG correlates across different streams, increasing the chances that at least *some* prefetches are issued for one of the streams.

Figures 4.10 and 4.11 show the accuracy of PC/DC and PC/DC/MG for the same set of prefetch degrees. In the vast majority of the cases the accuracy of PC/DC/MG is significantly higher than that of PC/DC. The accuracy of PC/DC/MG prefetches tends to remain stable or decrease only slightly with increasing prefetch degree. On the other hand, PC/DC's accuracy usually becomes worse as the prefetch degree increases.

Indeed PC/DC's "deep" prefetches (within a single stream) tend to have lower accuracy and lead to wasted bandwidth. For a single prefetch operation in PC/DC, the "tail" elements to be prefetched (e.g., the 15*th* and 16*th* prefetched blocks for a prefetching degree of 16) will refer to data that may not be referenced for a long time (recall that an indeterminate number of misses belonging to other streams may happen between two consecutive misses in the same stream, Section 4.3.2). The longer a prefetched block stays unreferenced in the cache, the higher the probability that it

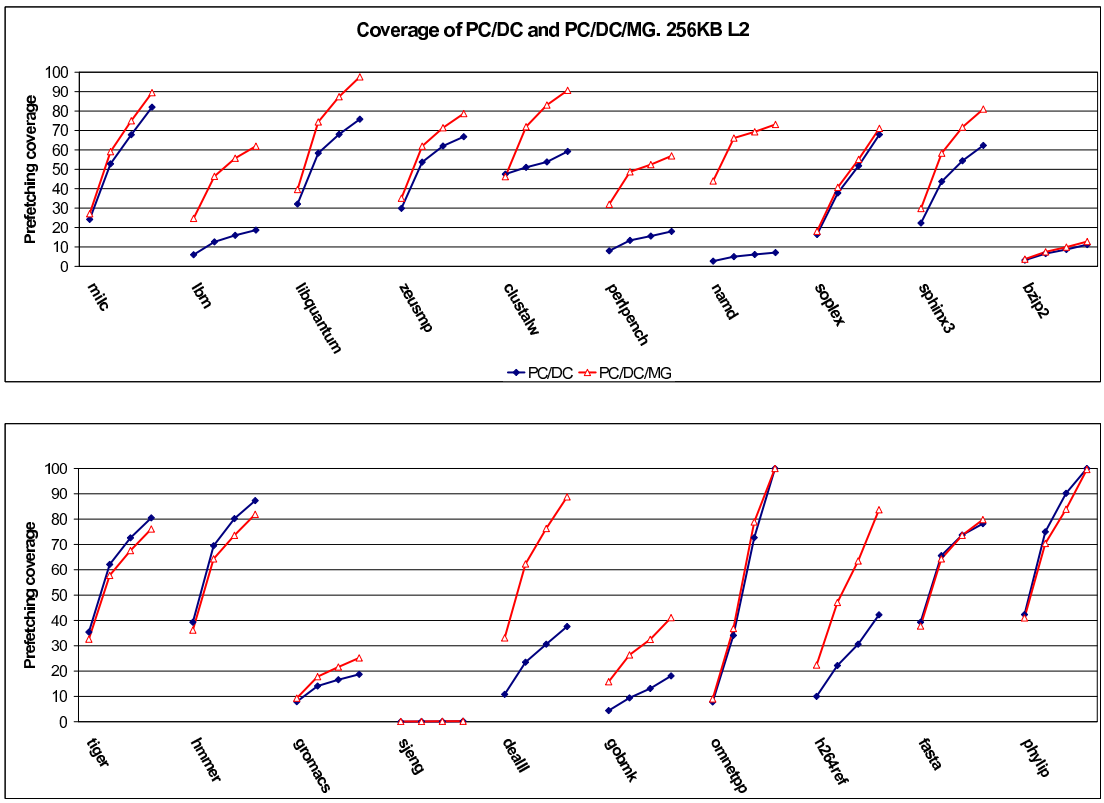


Figure 4.8: Prefetching coverage of PC/DC and PC/DC/MG. 256KB L2 cache.

will be replaced by other data. Therefore, the tail prefetches in PC/DC have higher probability of being replaced before being used at all.

By contrast, PC/DC/MG issues prefetches in a much more timely manner, increasing the chances of them being referenced before an eventual replacement. The tail prefetch elements of a PC/DC/MG activation will be used, in case of a correct prediction, much sooner; in the best case, the *n*th prefetched element will be used in exactly *n* misses.

We can correlate the accuracy and coverage against the performance results shown in Figures 4.6 and 4.7. It can easily be seen that where PC/DC/MG significantly outperforms PC/DC, this is due to an important increase in coverage, accuracy or indeed both. In *lbm*, *libquantum*, *dealII* and *gobmk*, PC/DC/MG shows much better accuracy and coverage, whereas in *perlbench* and *namd* the lesser accuracy of PC/DC/MG is compensated by a much larger coverage. Lastly, the PC/DC/MG results in *clustalw* are due to a higher coverage while at the same time maintaining roughly the same accuracy as PC/DC.

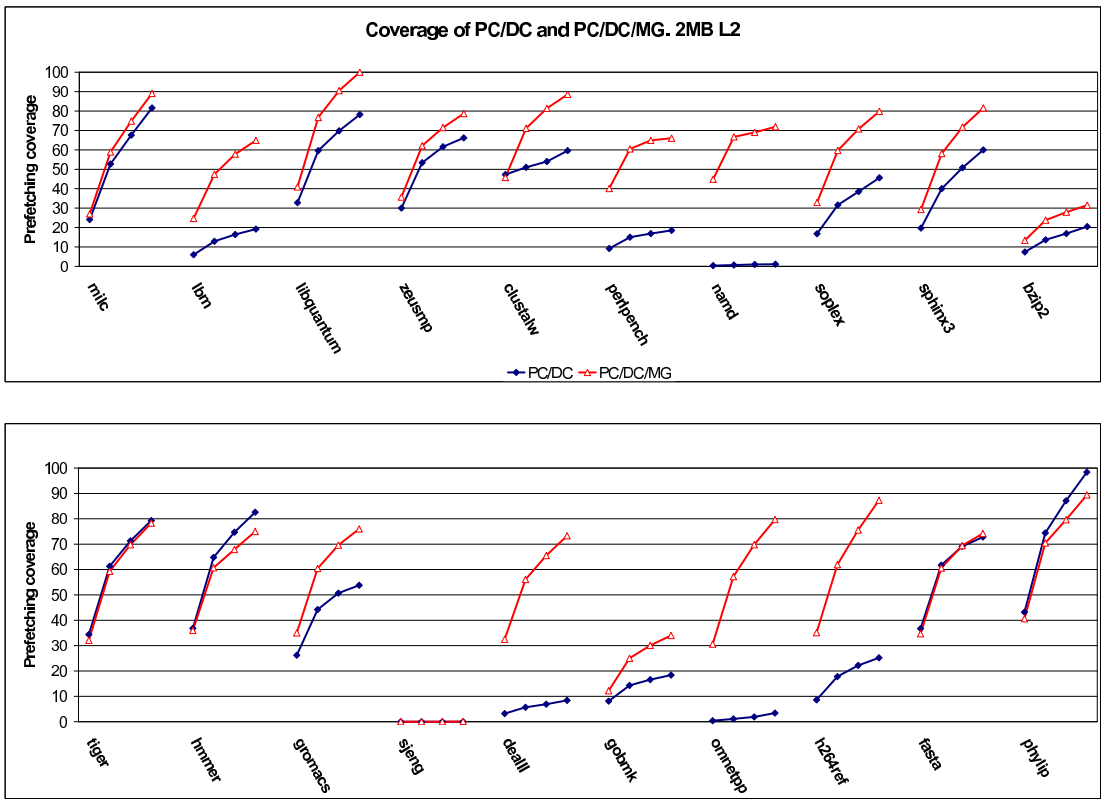


Figure 4.9: Prefetching coverage of PC/DC and PC/DC/MG. 2MB L2 cache.

4.4.3 PC Stream Prediction Accuracy

For a stream chaining prefetcher to work well, it is fundamental that it can predict accurately the next streams to be activated. Figures 4.12 and 4.13 show the accuracy of the predictions made by PC/DC/MG for 256KB and 2MB L2 cache sizes. We show the accuracy for various prediction windows w. A prediction is deemed correct if the stream predicted is activated in the next w misses. Naturally the accuracy increases as w increases. We consider prediction windows of up to 4 misses; any prediction that does not occur after that many misses is likely to be wrong, or of little use for our stream chaining approach.

In the 256KB L2 cache configuration the prediction accuracy of PC/DC/MG is already quite high for $w = 1$, with 15 out of 20 benchmarks above 80% and all except one above 65%. This indicates that the miss graph approach succeeds in capturing the most common transitions between PCs. The worst performer is *sjeng*, with an accuracy of 49% for $w = 1$. However, once we consider a window of two misses, the prediction accuracy for all benchmarks goes above 70%, with an average of 91%.

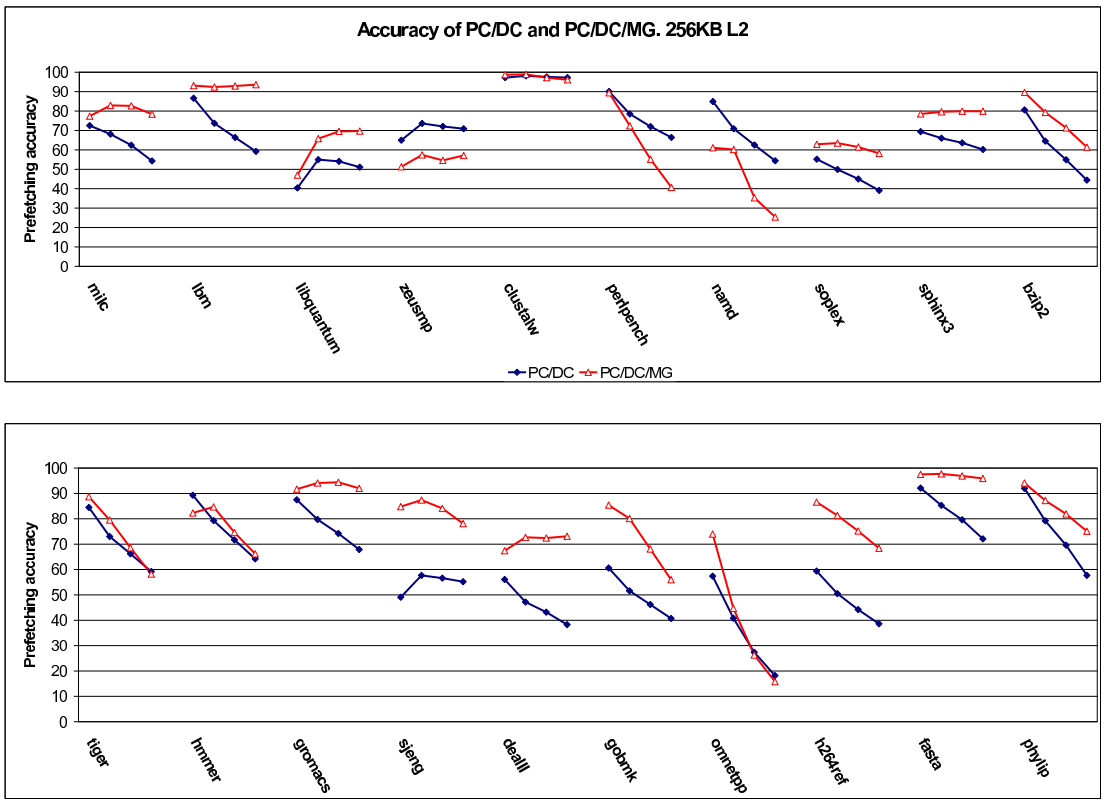


Figure 4.10: Prefetching accuracy of PC/DC and PC/DC/MG. 256KB L2 cache.

Further increasing the prediction window to $w = 4$ results in an overall accuracy of 95%, with no benchmark going below 80%. The results for the 2MB L2 cache are very similar to the 256KB configuration, with the only significant difference being an increase in prediction accuracy for *sjeng* (49% to 66% for $w = 1$).

Overall, the largest increase in stream prediction accuracy is from $w = 1$ to $w = 2$. One important factor that helps explain this behavior is the occurrence of spurious misses. As explained in 3.4.1.1, PC/DC/MG uses a saturation counter to avoid modifying too often the miss graph due to infrequent "noisy" misses. When we increase the stream prediction window from $w = 1$ to $w = 2$, we discount the effect that these spurious misses have in stream prediction by allotting space for them in the prediction window.

4.4.4 Miss Graph Characterization

Here we seek to gain more insight into the behavior of a stream chaining prefetcher by analyzing statistics about the miss graphs generated by PC/DC/MG. Table 4.2a char-

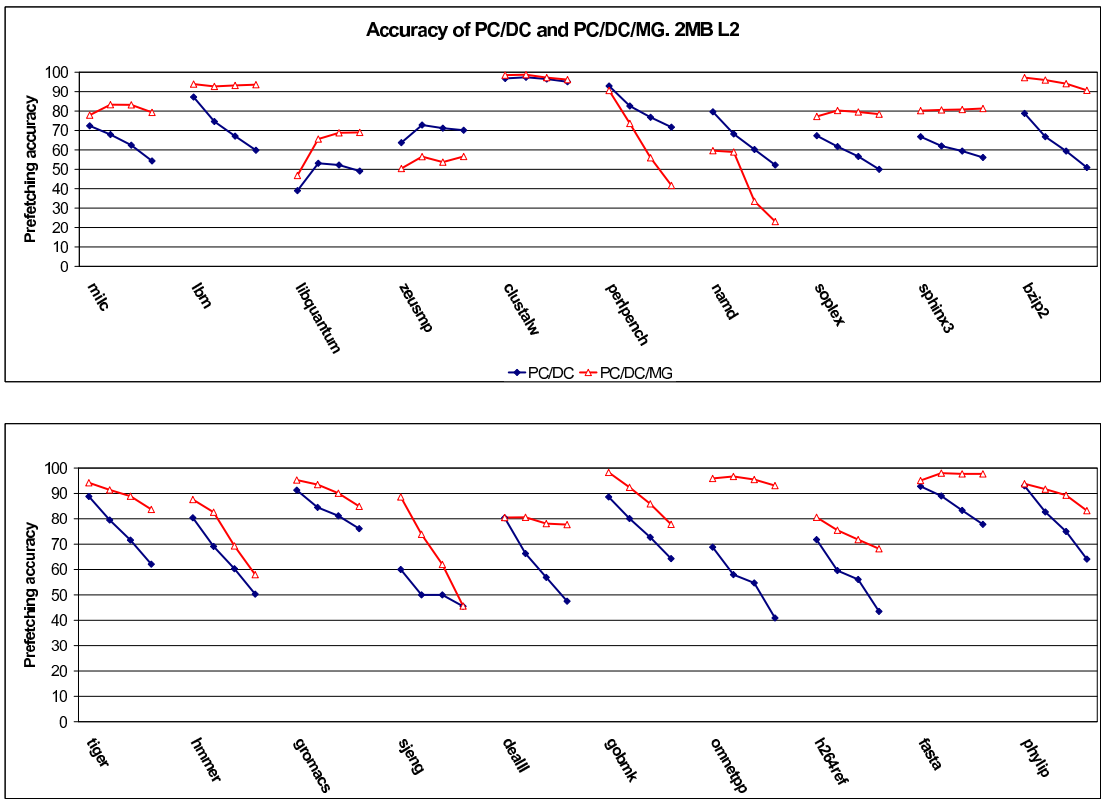


Figure 4.11: Prefetching accuracy of PC/DC and PC/DC/MG. 2MB L2 cache.

acterizes the graphs generated by the prefetcher. We obtained these statistics taking snapshots of the IT table (where the miss graph structure is built) every 1000 prefetch events. We also evaluated snapshots at equal intervals of L2 accesses and the results were very similar. At each snapshot we build the graph described by following the IT `NextIT` pointers. As explained in 3.4.1.2 we only consider those edges with a `Ctr` value of 3 or higher. From our experiments we have found that every snapshot contains a collection of several Connected Components (CC). These are subgraphs that are linked by edges.

In Table 4.2a *Unique Subgraphs* refers to the percentage of unique CC across the observed samples. *Snapshot* refers to the range and average number of nodes per snapshot (sample). *CC* refers to the range and average number of nodes per connected component in each graph.

The *Unique Subgraphs* column allows us to measure how stable the miss graphs are across the execution of the program. Stable miss graphs are key for the success of a stream chaining prefetcher. If the graphs are unstable or change too often the prefetcher will not have enough leverage to issue good, timely prefetches. Because comparing

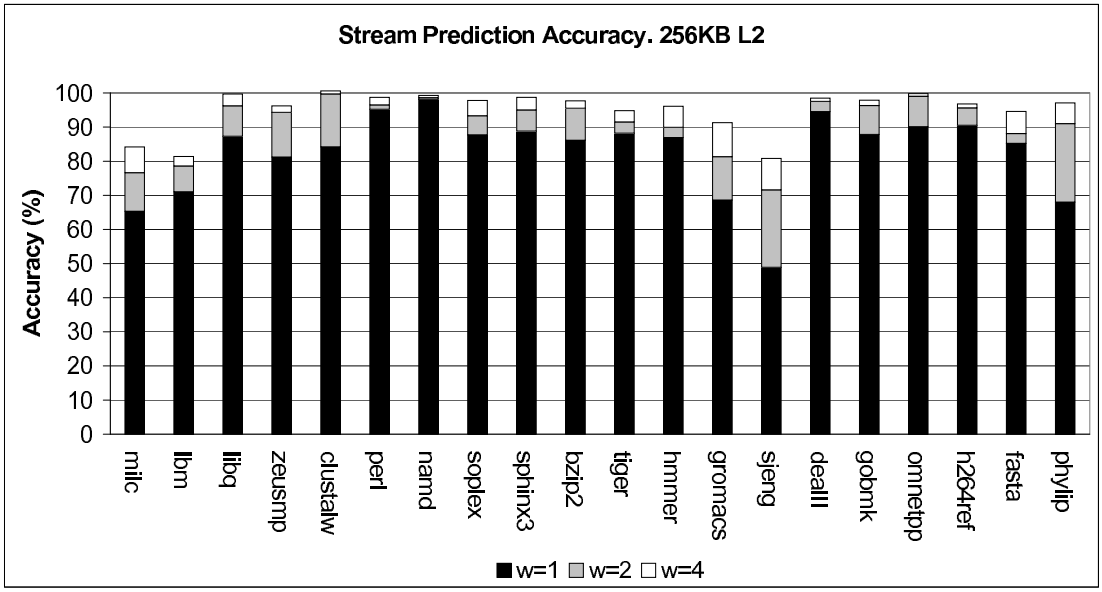


Figure 4.12: PC/DC/MG accuracy in predicting the next PC to appear in the next *w* misses. 256KB L2 Cache.

the CC subgraphs across the very large number of snapshots taken is prohibitively expensive, we use a comparison window of 30 snapshots ahead. This window size was determined empirically, and larger windows do not affect the results significantly. We classify every CC subgraph from every snapshot as either *similar* to at least one other CC in one other snapshot in the comparison window, or *unique*, meaning that there is no other similar CC in the window. We deem two CC X and Y similar if X is a subgraph of Y and X's nodes correspond to no less than 75% of Y's nodes, or if Y is a subgraph of X and Y's nodes correspond to no less than 75% of X's nodes. Note that by this definition, two CC that are exactly the same will be classed as "similar".

In columns 2 and 3 of Table 4.2 we show the fraction of unique CC subgraphs for 256KB and 2MB L2 cache sizes. The fraction of unique subgraphs seen is very small for most benchmarks. For a 256KB L2, only 11.8% of the observed CC are unique on average, with just 4 out of 20 benchmarks showing more than 15% of different subgraphs. For a 2MB L2, 16.5% of the CC are unique, with 7 benchmarks having more than 15% of unique subgraphs. The higher percentage of different subgraphs seen in the 2MB L2 configuration can be explained by the buffer effect a bigger cache has on the global miss stream, capturing more of the more frequent request and increasing the entropy of the misses. In any case, both configurations show a high number of similar subgraphs. This suggests that the graphs constructed by PC/DC/MG are stable

Benchmark	Unique Subgraphs (%)		Nodes			
			Snapshot		CC	
	256KB	2MB	256KB	2MB	256KB	2MB
milc	4.7	3.0	[2, 15] 7.7	[2, 15] 7.9	[1, 7] 3.6	[1, 7] 3.5
lbm	22	12	[2, 20] 7.9	[2, 18] 6.7	[2, 18] 3.7	[2, 18] 4.2
libq	0.8	1.3	[2, 23] 19	[1, 24] 21	[1, 18] 7.0	[1, 18] 7.4
zeusmp	11	10	[2, 18] 11	[2, 15] 8.8	[2, 9] 4.4	[2, 10] 4.1
clustalw	1.1	7.2	[3, 10] 9.3	[2, 9] 6.7	[2, 10] 8.2	[1, 9] 6.6
perl	11	7.1	[1, 16] 8.6	[2, 18] 9.9	[1, 9] 3.3	[1, 9] 3.6
namd	21	23	[2, 8] 5.8	[2, 8] 6.0	[2, 8] 5.0	[2, 8] 5.8
soplex	2.8	12	[1, 30] 12	[1, 11] 6.2	[1, 10] 3.6	[1, 6] 3.0
sphinx3	11	9.6	[4, 16] 13	[2, 11] 7.1	[1, 15] 5.7	[1, 5] 3.4
bzip2	5.6	15	[1, 38] 20	[1, 25] 13	[1, 9] 3.8	[1, 7] 3.7
tiger	5.4	6.7	[7, 41] 30	[6, 34] 23	[1, 18] 4.2	[1, 14] 3.8
hmmer	12	9.4	[15, 50] 38	[13, 36] 28	[1, 33] 5.4	[1, 26] 4.5
gromacs	15	42	[2, 25] 13	[4, 13] 9.8	[1, 12] 3.6	[1, 7] 4.4
sjeng	45	29	[2, 13] 7.7	[2, 14] 8.4	[2, 13] 5.1	[2, 12] 5.2
dealII	6.4	43	[1, 25] 14	[1, 9] 4.7	[1, 11] 4.1	[1, 7] 3.1
gobmk	20	31	[1, 10] 5.2	[2, 13] 9.3	[1, 5] 3.4	[1, 9] 4.5
omnetpp	6.2	1.6	[1, 9] 3.7	[3, 14] 4.8	[1, 4] 3.0	[1, 3] 1.8
h264ref	14	22	[1, 7] 4.6	[1, 9] 8.0	[1, 4] 2.9	[1, 4] 2.8
fasta	8.9	8.9	[8, 19] 15	[8, 17] 15	[1, 16] 5.4	[1, 9] 3.9
phylip	13	37	[8, 36] 26	[8, 14] 11	[1, 24] 5.1	[1, 10] 4.6
Average	11.8	16.5	13.5	10.7	4.5	4.1

(a)

Benchmark	GHB hops			
	PC/DC		PC/DC/MG	
	256KB	2MB	256KB	2MB
milc	43	43	55	55
lbm	7.7	8.8	9.2	11
libq	52	57	49	51
zeusmp	31	34	371	384
clustalw	9.5	11.6	31	31
perl	19	25	94	143
namd	37	38	353	352
soplex	79	55	101	60
sphinx3	118	131	123	137
bzip2	202	104	215	109
tiger	27	31	109	60
hmmer	19	57	131	158
gromacs	76	44	86	70
sjeng	64	95	148	332
dealII	85	94	116	97
gobmk	124	106	136	116
omnetpp	306	55	313	67
h264ref	18	47	30	63
fasta	24	12	28	14
phylip	52	49	76	44
Average	69	54	128	117

(b)

Table 4.2: PC/DC/MG: miss graphs statistics (a) and GHB hop counts (b).

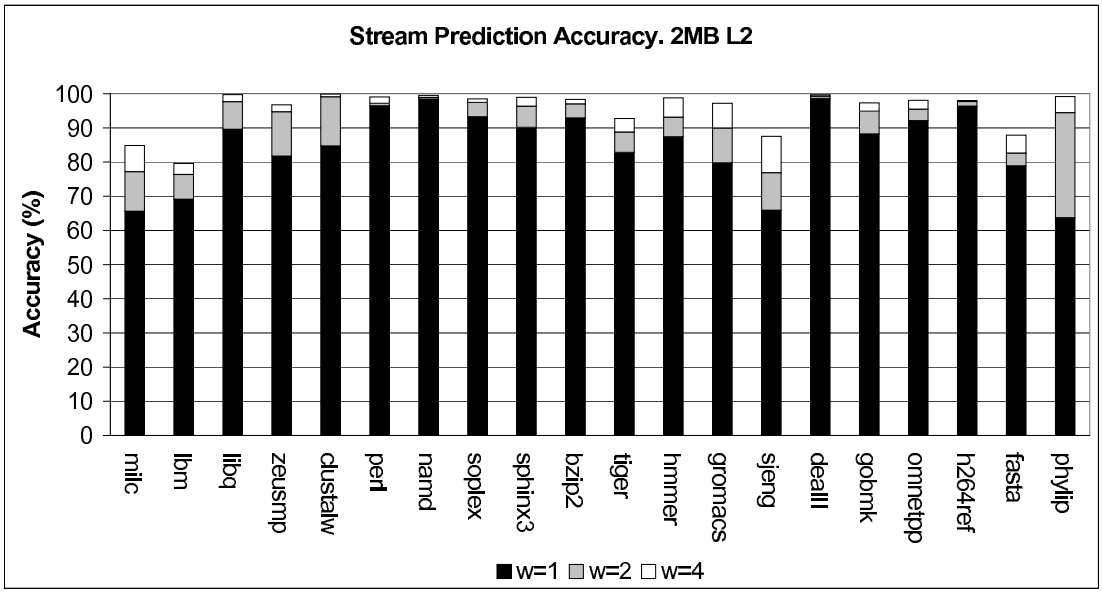


Figure 4.13: PC/DC/MG accuracy in predicting the next PC to appear in the next *w* misses. 2MB L2 Cache.

and do not change much over time. These results are in line with the good next stream prediction accuracy shown in Section 4.4.3.

Table 4.2a also shows - in columns 4 to 7 - the average and range of number of nodes both per snapshot and per CC. The number of nodes per snapshot, and specially, per CC is not so large that managing the graphs - and, thus, operating the stream chaining mechanism - becomes too expensive.

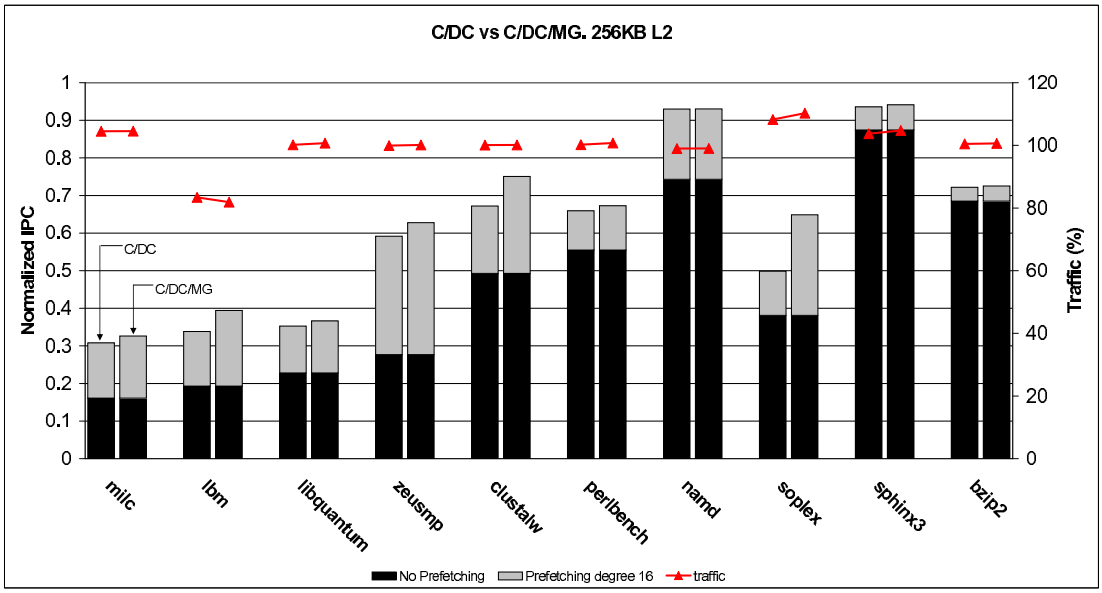
Table 4.2b shows the average number of GHB lookups (“hops”) required by each prefetcher to establish delta correlations on a miss event, for a prefetch degree of 16. The number of GHB hops is larger in PC/DC/MG than in PC/DC. This can be explained by the fact that PC/DC/MG visits several streams everytime it is activated. However, this number of hops is not large enough to hinder the performance of the prefetcher and is within the time constraints offered by a typical L2 cache.

4.5 C/DC/MG

4.5.1 Performance and Traffic

Figures 4.14 and 4.15 show the performance improvement and traffic variation for C/DC and C/DC/MG. As with the results for PC/DC/MG, we plot performance results

normalized to the IPC obtained with an ideal L2 cache. We use 64KB CZones in both prefetchers, since by experimentation we have found that they provide the best all-around performance in our benchmark set.



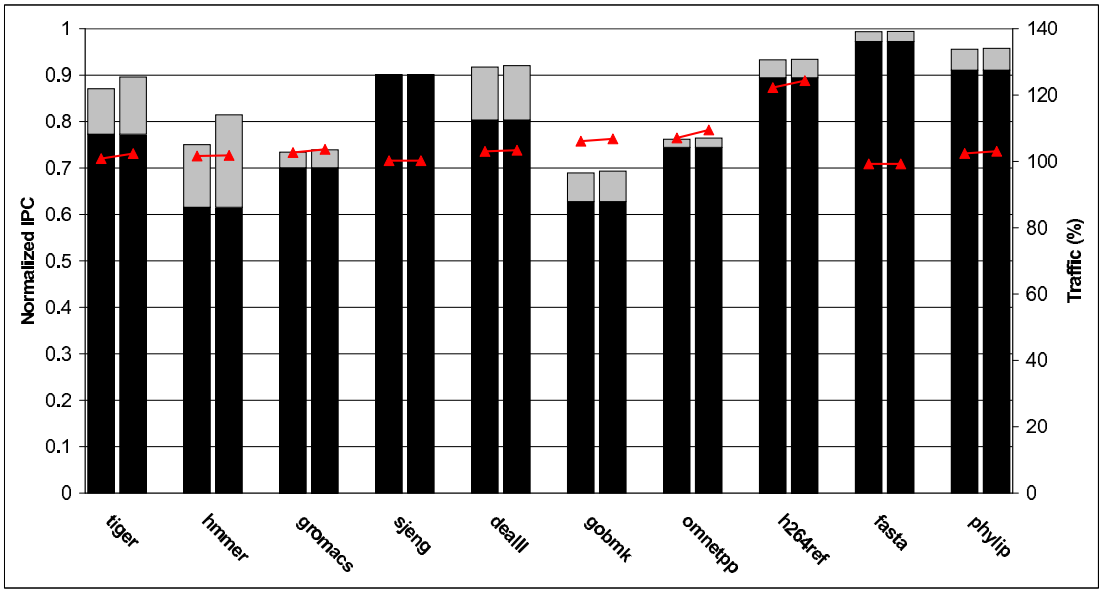


Figure 4.14: C/DC (left bar) and C/DC/MG (right bar) performance and traffic. 256KB L2 cache.

For a L2 cache size of 256KB, C/DC/MG improves significantly the performance of C/DC in 9 of the 20 benchmarks. The greatest improvement happens in the benchmark *soplex*, where C/DC achieves a normalized IPC of 0.50 and C/DC/MG records a normalized IPC of 0.65, an improvement of 30.1%. Other benchmarks that benefit

significantly from C/DC/MG are *lbm* (0.34 to 0.39 normalized IPC, a 16.6% improvement), *clustalw* (0.67 to 0.75 normalized IPC, a 11.6% improvement) *hmmer* (0.75 to 0.81 normalized IPC, a 8.5% improvement). The lowest improvement that is above 1% corresponds to the *perlbench* benchmark (2.1%). The rest of the benchmarks do not show any significant improvement (i.e., above 1%) by using C/DC/MG instead of C/DC. However, and unlike PC/DC/MG, C/DC/MG never degrades the performance compared to its non-stream chaining algorithm.

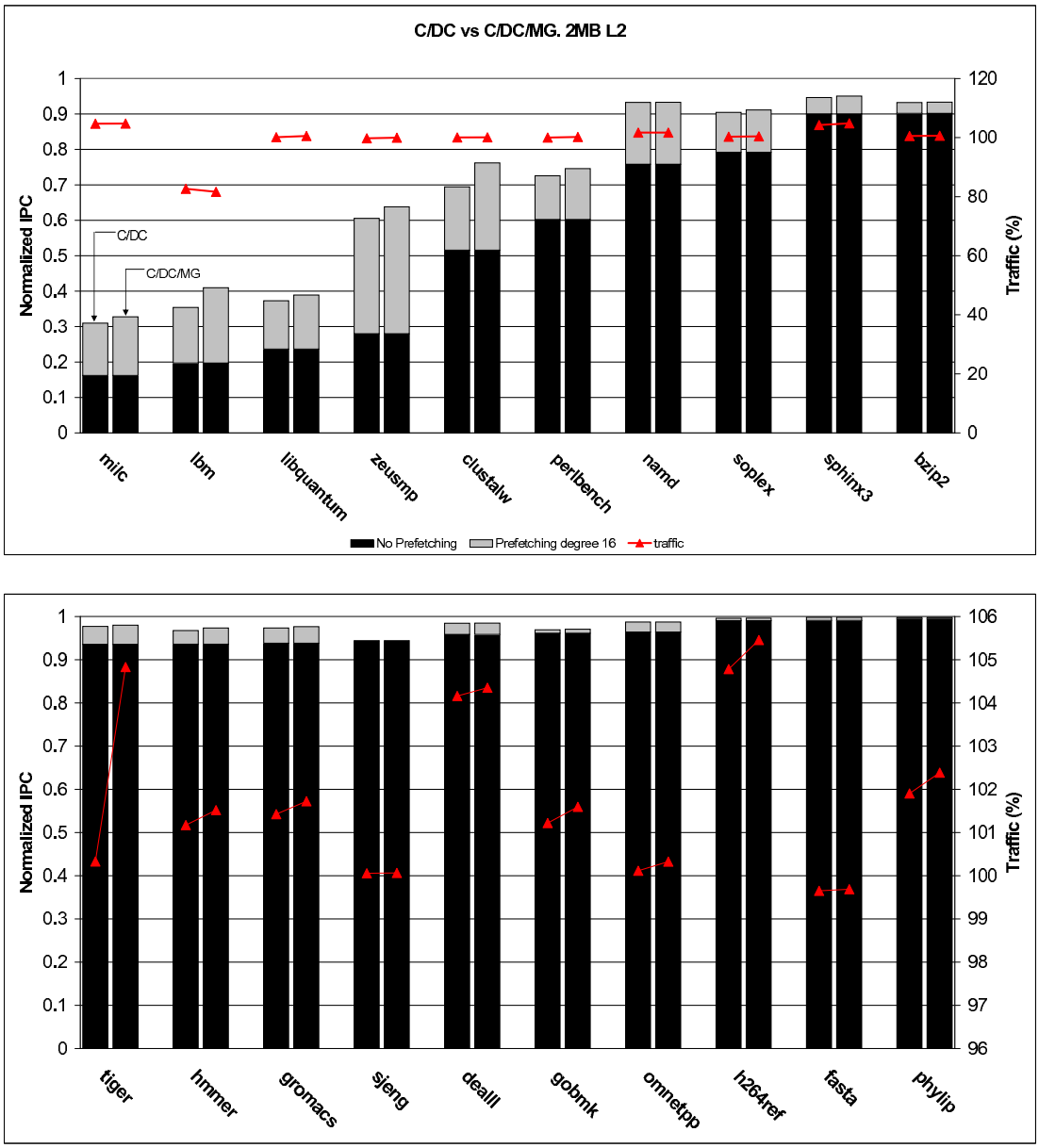


Figure 4.15: C/DC (left bar) and C/DC/MG (right bar) performance and traffic. 2MB L2 cache.

For a 2MB L2 cache size, C/DC/MG improves significantly the performance of C/DC in 6 of the 20 benchmarks. The highest improvement is in the benchmark *lbm*, where C/DC/MG improves the normalized IPC from 0.35 to 0.41, a 15.6% increase in performance. The next two greatest improvements are in *clustalw* (0.69 to 0.76, a 9.7% improvement) and *milc* (0.31 to 0.33, a 5.8% improvement). The lowest improvement that is above 1% happens in *perlbench* (2.8%). Similarly to the 256KB L2 case, there is no performance penalty for using C/DC/MG. As with PC/DC/MG, the 2MB L2 configuration pushes more benchmarks closer to the ideal L2 performance, so obtaining any performance improvement in these cases is difficult. For example, 3 benchmarks that benefited from C/DC/MG in the 256KB configuration (*soplex*, *hmmer* and *tiger*) surpass the 0.9 normalized IPC mark with C/DC prefetching in the 2MB L2 configuration.

Both C/DC and C/DC/MG show low traffic increase on most benchmarks. In the 256KB configuration the benchmark with the highest traffic increase is *h264ref*, with 122% and 124% of the original traffic with no prefetching for C/DC and C/DCMG respectively. For a 2MB L2 cache the benchmark that shows the highest traffic increase is *milc*, with 105% (C/DC and C/DC/MG) of the original traffic. As with PC/DC and PC/DC/MG, we observe a significant decrease in bus traffic with the benchmark *lbm*. This is also due to a strong reduction in the L2 writeback traffic.

Overall, the results of C/DC/MG show that, in some benchmarks, significant improvements can be achieved with stream chaining, even with a spatial localization algorithm. Spatial localization algorithms tend to group misses in such a way that there are fewer miss stream transitions for a given period of time. This is due to spatial locality in the program (i.e., if a program references an area of memory, it will tend to reference as well nearby areas). Fewer stream transitions means shorter miss graphs and, therefore, less opportunities for stream chaining to improve the timeliness of the prefetch stream.

4.5.2 Coverage and Accuracy

Figures 4.16 and 4.17 show the coverage of C/DC and C/DC/MG for 256KB and 2MB L2 caches. In all benchmarks C/DC/MG's coverage is equal to or larger than C/DC. For a 256KB cache, the most significant increases in coverage (measured at prefetching degree 16) are in the benchmarks *hmmer* (60.4% to 80.9%), *clustalw* (54.7% to 70.6%) and *soplex* (54.6% to 67.1%). In the 2MB L2 configuration, the largest cover-

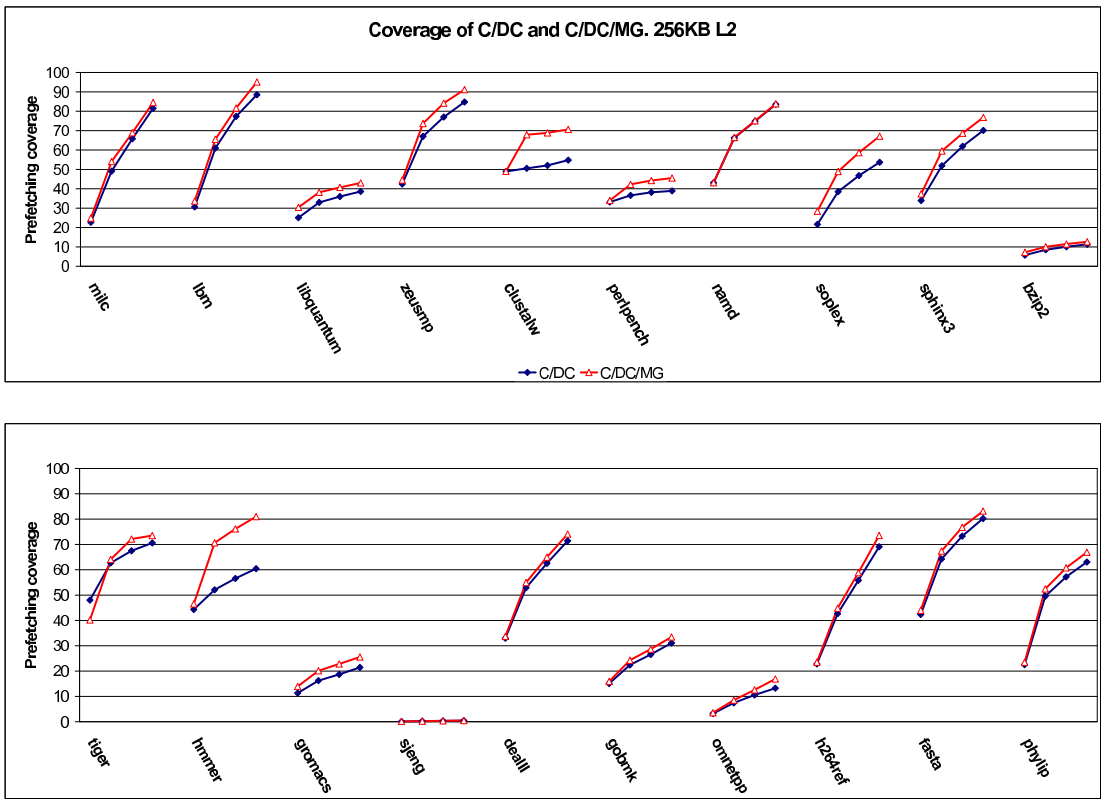


Figure 4.16: Prefetching coverage of C/DC and C/DC/MG. 256KB L2 cache.

age increases happen in the *clustalw* (55.1% to 69.3%), *hmmer* (59.7% to 72.2%) and *tiger* (70.4% to 78.8%) benchmarks.

Overall the results show not only that the coverage of C/DC/MG is generally higher than that of C/DC, but also that this increase becomes larger as the prefetch degree increases. As with PC/DC/MG, this is due to increased opportunities for prefetching: some CZones might not have enough miss data to predict more than a small number of misses, but with the Stream Chaining strategy we can correlate over several CZones. This increases the number of prefetches issued per prefetcher activation, therefore achieving a higher coverage.

In Figures 4.18 and 4.19 we show the accuracy of both prefetchers for 256KB and 2MB L2 caches. Overall, the accuracy of C/DC and C/DC/MG is roughly the same, with a few exceptions with the 256KB L2 cache. For this configuration C/DC/MG suffers significant accuracy degradation in the benchmark *h264ref* (96.4% to 79.2%), and moderate to small degradation in the benchmarks *lbm* (80% to 74.4%) and *hmmer* (95.3% to 91.1%). However, two of these benchmarks (*h264ref* and *hmmer*) already perform close to an ideal L2 without prefetching. All these benchmarks, however,

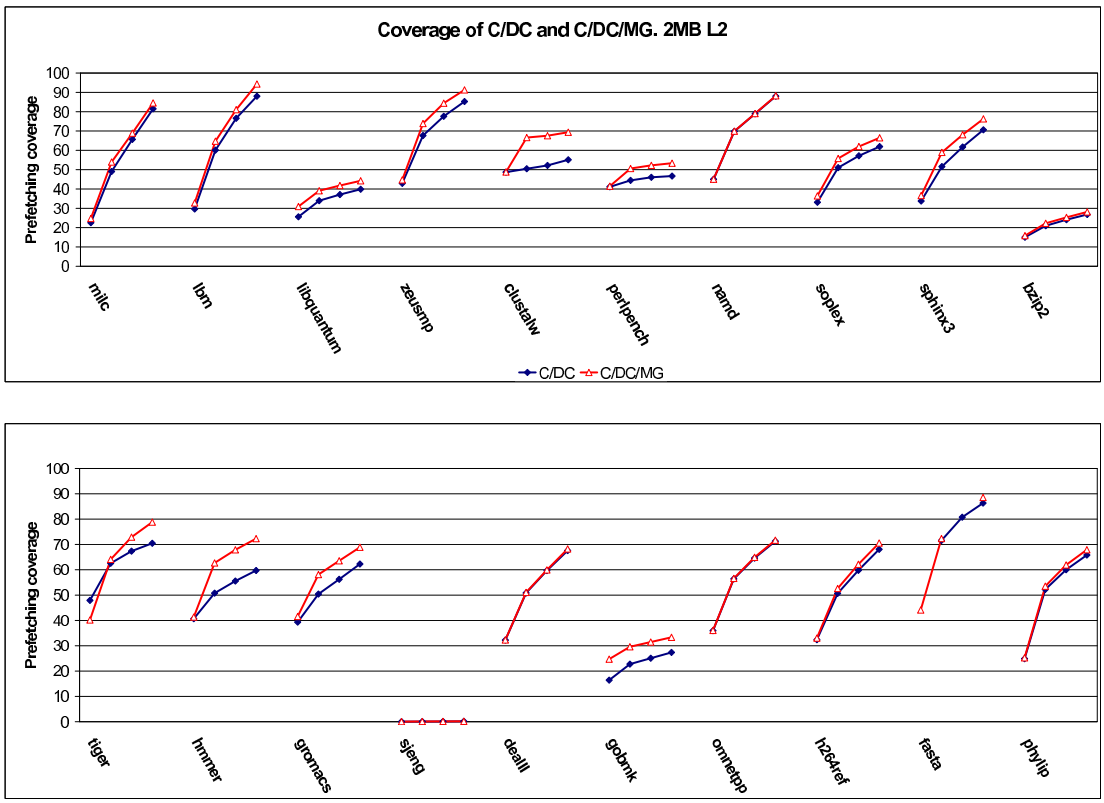


Figure 4.17: Prefetching coverage of C/DC and C/DC/MG. 2MB L2 cache.

perform better with C/DC/MG than C/DC, so this decrease in accuracy is compensated by higher coverage and better prefetch timeliness.

4.5.3 CZone Transition Prediction Accuracy

In Figures 4.20 and 4.21 we show the accuracy of C/DC/MG in predicting the next CZone to be activated. This data refers only to transition between different CZones; the accuracy of predicting consecutive misses to the same CZone is analyzed in Section 4.5.4.

As with PC/DC/MG, we establish a window w of 1, 2 or 4 misses. In this context, this window size refers to the number of misses to different CZones.

For a window $w = 1$, 10 out of the 20 benchmarks present prediction accuracies above 70% for a 256KB L2. With the exception of *lbm* and *sphinx3*, the prediction results show little variation with respect to the window size. For $w = 4$, 12 benchmarks show accuracies over 70%, 4 between 40% and 70% and the remaining 4 have accuracies lower than 40%. For a 2MB cache, 9 out of the 20 benchmarks have prediction

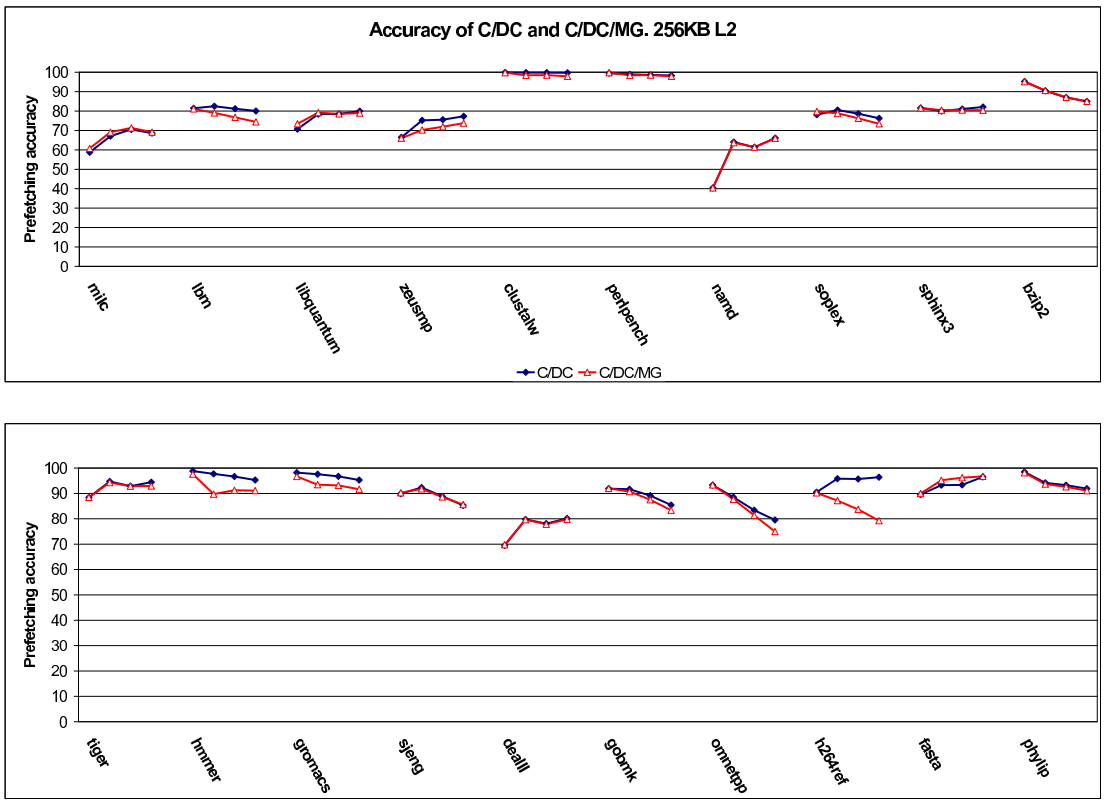


Figure 4.18: Prefetching accuracy of C/DC and C/DC/MG. 256KB L2 cache.

accuracies above 70%. Moving to a larger window size $w = 4$, 13 benchmarks show accuracies over 70%, 4 present accuracies in the 40% - 70% range [1] and the remaining 3 show accuracies less than 40%. Like with a 256KB cache, there is little variability when changing the window size with the exception of *lbm* and *sphinx3*.

Overall the prediction results show that CZone transitions are more difficult to predict than PC transitions. This is expected, since whereas the PC miss graphs are due to a inherently structured source (the program), the CZone miss graphs are highly dependent on the spatial patterns of the miss address stream.

4.5.4 CZone Repetition Prediction Accuracy

Figures 4.22 and 4.23 show the accuracy of the C/DC/MG in predicting how many consecutive misses will each CZone receive (CZone repetitions). Data is given as the frequency that the predictor is right plus or minus $d = \{0, 1, 2\}$ misses. The data for $d = 0$ shows the number of occasions C/DC/MG predicted the exact number of

[1] We included *libquantum* in this group, with a $w = 4$ accuracy of 39.82%

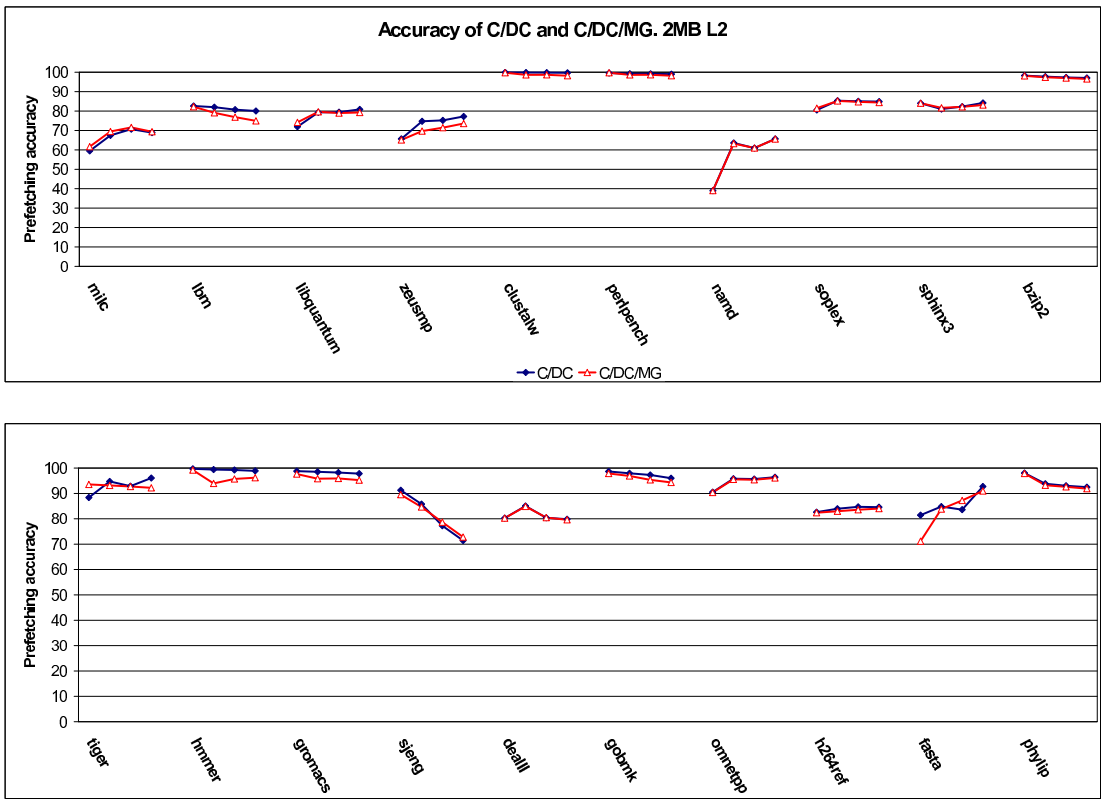


Figure 4.19: Prefetching accuracy of C/DC and C/DC/MG. 2MB L2 cache.

repetitions, $d = 1$ represents the data for when the prediction was off by plus or minus 1 miss and similarly $d = 2$ represents the data for predictions that are off by 2 misses.

For both cache sizes the C/DC/MG makes exact predictions ($d = 0$) more than 70% of the time in 12 out of the 20 benchmarks. For a cache size of 256KB and $d = 1$, 18 benchmarks show prediction accuracies above 70% and 13 of them have accuracies over 90%. For a 2MB L2 cache and $d = 1$, 16 benchmarks have prediction accuracies above 70% and 12 of them above 90%. The data for $d = 2$ shows a smaller increment in accuracy for both cache sizes, indicating that most C/DC/MG predictions are either right within one miss (the majority), or off by more than two misses.

As with the CZone transition predictions, the results are very similar for 256KB and 2MB caches, suggesting that the mechanism is robust enough to be independent of the cache size.

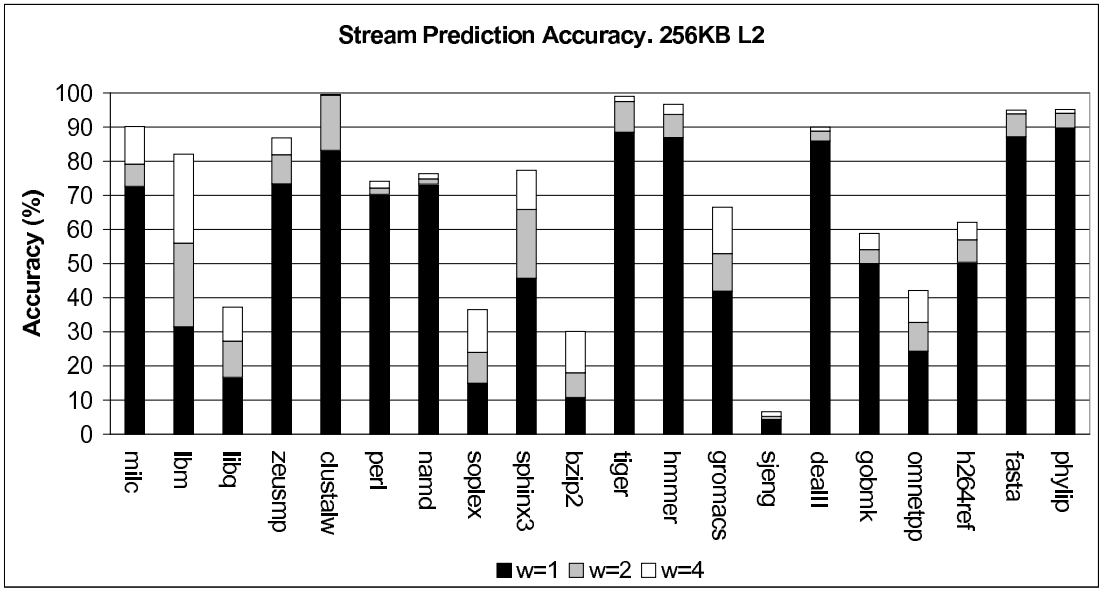


Figure 4.20: C/DC/MG accuracy in predicting the next CZone to appear in the next *w* misses. 256KB L2 Cache.

4.5.5 Miss Graph Characterization

Table 4.3 characterizes the miss graphs used by C/DC/MG. As with PC/DC/MG, we calculate the percentage of unique graphs [2] (columns 2 and 3), the average size of the graphs on each snapshot (columns 4 and 5), the average size of the Connected Components (CC) of each snapshot (columns 6 and 7) and the average number of GHB hops for C/DC and C/DC/MG (Table 4.3b).

Overall, the results show that with C/DC/MG the percentage of unique graphs is higher on average than with PC/DC/MG. Comparing the size of the graphs with the CZone prediction accuracy, we can see that all the benchmarks that show poor accuracies (i.e., *libquantum*, *soplex*, *bzip2*, *sjeng* and *omnetpp*, for a 256KB L2) have also a high proportion of unique graphs. This indicates that in these benchmarks there is high variability in the structure of the graphs, and this affects their predictability. On the other hand, some benchmarks with a high percentage of unique graphs (*perlbench*, *tiger*, *fasta* and *phylip*) show good CZone prediction accuracies.

The average number of nodes in each graph snapshot is much higher than with PC/DC/MG. On the other hand, the average size of each CC is on average less than in PC/DC/MG. This means that C/DC/MG captures more different transitions between

[2] We use the same procedure described in Section 4.4.4.

Benchmark	Unique Subgraphs (%)		Nodes				GHB hops			
			Snapshot		CC		C/DC		C/DC/MG	
	256KB	2MB	256KB	2MB	256KB	2MB	256KB	2MB	256KB	2MB
milc	5.6	5.9	[12, 424] 336	[12, 423] 332	[2, 330] 7.6	[2, 338] 7.9	35	35	38	37
lbm	1.9	2.1	[4, 339] 271	[4, 366] 265	[2, 42] 3.3	[2, 40] 3.4	56	52	72	80
libq	25	26	[8, 375] 236	[2, 373] 211	[2, 95] 3.1	[2, 72] 2.9	8.8	9	30	22
zeusmp	2.8	2.2	[10, 315] 207	[12, 383] 249	[2, 42] 3.2	[2, 37] 3.1	33	36	63	70
clustalw	2.9	3.4	[3, 139] 124	[3, 137] 121	[2, 17] 3.4	[2, 23] 3.7	7.7	9.5	30	43
perl	26	21	[37, 159] 115	[35, 86] 64	[2, 99] 5.1	[2, 40] 2.7	4.7	4.1	5.8	5.9
namd	16	19	[40, 122] 93	[44, 109] 89	[2, 22] 4.0	[2, 35] 5.0	39	32	42	33
soplex	19	11	[2, 75] 19	[2, 129] 73	[2, 34] 2.8	[2, 31] 2.9	20	18	54	20
sphinx3	9.8	13	[8, 77] 57	[8, 82] 57	[2, 21] 2.8	[2, 21] 3.5	14	16	25	28
bzip2	22	30	[7, 30] 16	[7, 35] 20	[2, 12] 2.4	[2, 19] 3.0	8.8	8.8	13	13
tiger	13	13	[5, 286] 95	[5, 83] 58	[2, 112] 2.5	[2, 67] 6.4	39	30	63	46
hmmer	8.4	15	[5, 105] 62	[5, 101] 65	[2, 44] 4.6	[2, 53] 6.9	16	11	24	18
gromacs	16	12	[8, 42] 29	[8, 77] 57	[2, 17] 2.8	[2, 23] 3.0	68	33	90	44
sjeng	46	56	[4, 115] 43	[10, 90] 42	[2, 5] 2.1	[2, 6] 2.1	3.3	3.2	3.3	3.2
dealII	14	17	[8, 33] 24	[6, 120] 73	[2, 13] 3.1	[2, 28] 3.1	59	44	62	45
gobmk	33	38	[6, 40] 27	[7, 37] 24	[2, 15] 2.9	[2, 12] 2.8	48	25	50	25
omnetpp	13	17	[82, 151] 125	[62, 307] 173	[2, 28] 3.3	[2, 70] 2.6	11	60	13	64
h264ref	46	32	[6, 24] 14	[11, 71] 44	[2, 9] 2.5	[2, 20] 2.9	27	39	38	50
fasta	23	11	[6, 43] 25	[6, 72] 40	[2, 9] 3.3	[2, 8] 3.3	40	51	41	51
phylip	25	25	[6, 43] 23	[8, 53] 34	[2, 20] 3.2	[2, 23] 3.2	70	91	71	109
Average	18.4	18.5	97.0	105.3	3.4	3.7	30	30	41	40

(a) (b)

Table 4.3: C/DC/MG: miss graphs statistics (a) and GHB hop counts (b).

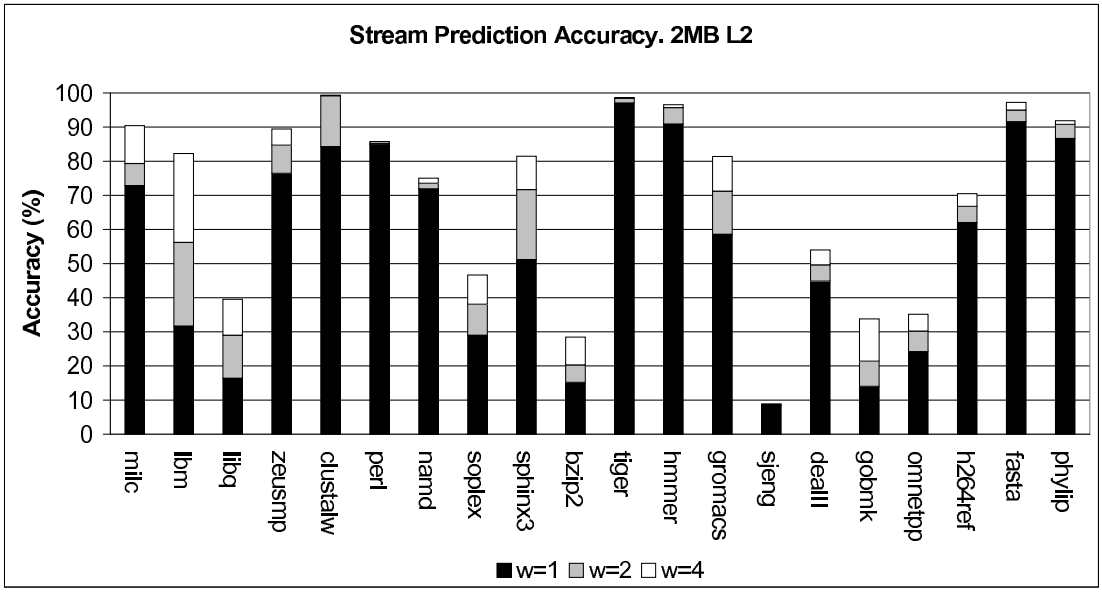


Figure 4.21: C/DC/MG accuracy in predicting the next CZone to appear in the next *w* misses. 2048KB L2 Cache.

streams than PC/DC/MG, but the individual transition chains are on average shorter. This is reflected as well on the number of GHB hops (Table 4.3b), close on average to the GHB hop count of C/DC.

4.6 Comparison of PC/DC/MG, C/DC/MG and G/DC

Having analyzed the performance of both our Stream Chaining prefetchers against their non-chaining counterparts, we now compare them against G/DC. G/DC is a simple prefetcher that does not perform any localization and instead operates on the global miss stream. As all the other prefetchers analyzed in this chapter, it uses Delta Correlation for detecting memory patterns and predicting future memory accesses.

Since G/DC does not perform localization, all the prefetches it generates are based on the global miss stream and therefore are expected to be very timely. On the other hand, because of the lack of localization mechanisms, it will not be able to detect interleaved memory patterns that are typical of complex benchmarks. By contrast, our Stream Chaining prefetchers do support localization and at the same time they promote a timely dispatch of prefetches.

Figures 4.24 and 4.25 show the performance and traffic results of PC/DC/MG, C/DC/MG and G/DC for 256KB and 2MB L2 caches. The first thing to note is that no

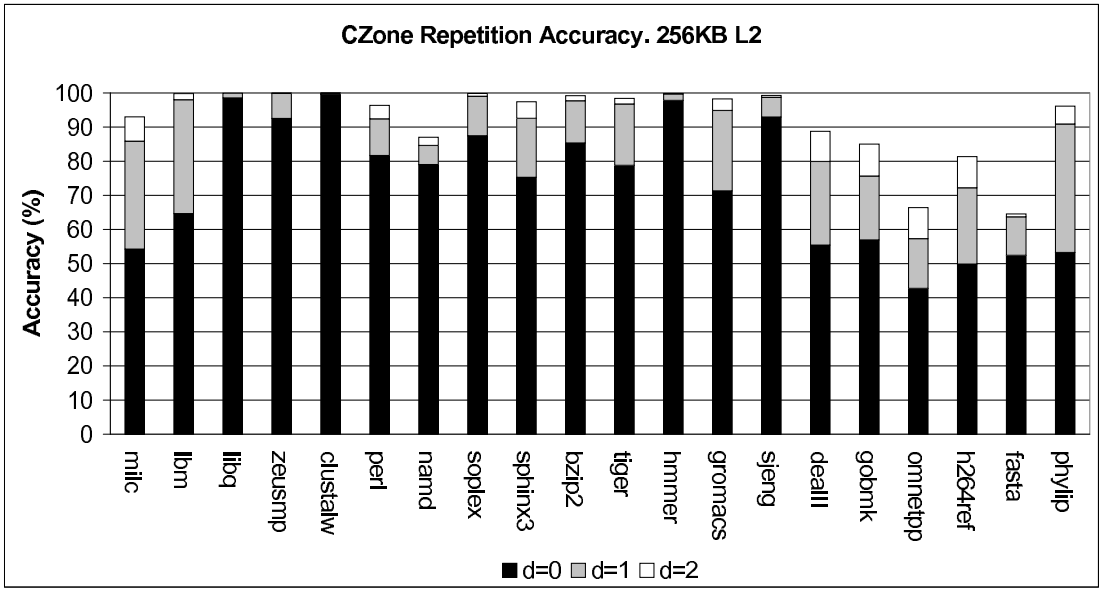


Figure 4.22: CZone repetition accuracy. 256KB L2 cache.

prefetcher gives the best performance in all benchmarks. This is expected, as different benchmarks show different memory patterns that are captured best with different localization schemes (or no localization).

For a 256KB L2 cache, the Stream Chaining prefetchers perform better than G/DC in 11 of the 20 benchmarks: *milc, lbm, libquantum, zeusmp, clustalw, perlbench, soplex, sphinx3, gromacs, gobmk* and *omnetpp*. G/DC obtains better results in 2 benchmarks: *namd* (1.9% over C/DC/MG) and *hmmer* (5.7% over PC/DC/MG). Two benchmarks show significant and roughly equal improvements with the Stream Chaining and G/DC prefetchers: *tiger* (19.2% over non-prefetching baseline for PC/DC/MG, 19.6% for G/DC) and *dealII* (16.6% for PC/DC/MG and 14.1% for G/DC). Lastly, the 5 remaining benchmarks (*bzip2, sjeng, h264ref, fasta* and *phylip*) show little or no improvement in performance with prefetching, be it Stream Chaining or G/DC.

As expected, the results for a 2MB L2 cache show less benefits from prefetching. In 7 of the 20 benchmarks our Stream Chaining Prefetchers obtained better performance improvements than G/DC: *milc, lbm, libquantum, zeusmp, clustalw, soplex* and *sphinx3*. G/DC performed better in two benchmarks: *perlbench* and *namd*. Lastly, the remaining 10 benchmarks (*tiger, hmmer, gromacs, sjeng, dealII, gobmk, omnetpp, h264ref, fasta* and *phylip*) showed small or no improvements with prefetching.

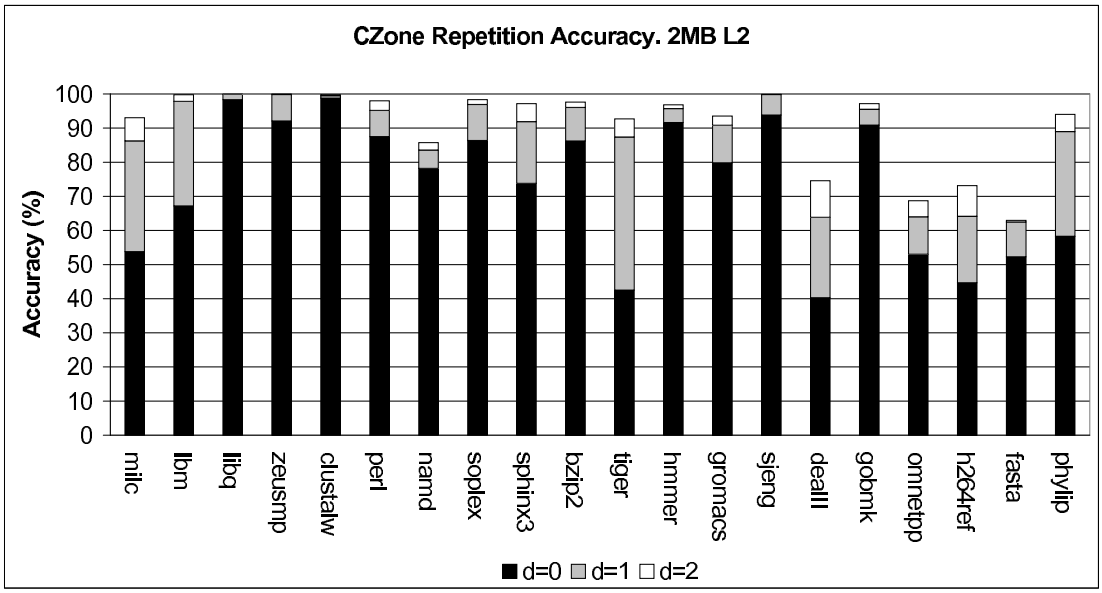


Figure 4.23: CZone repetition accuracy. 2MB L2 cache.

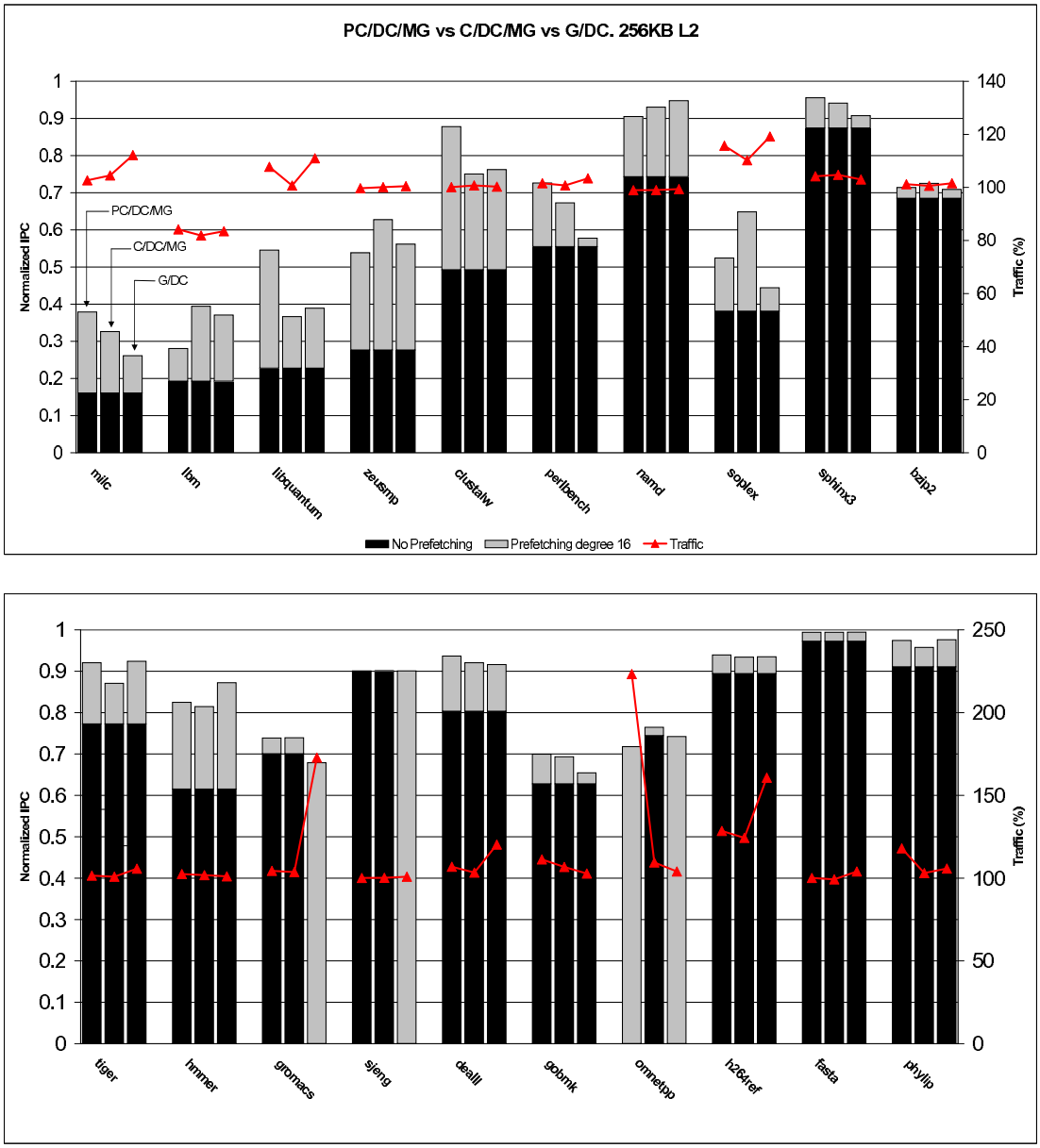


Figure 4.24: Comparison of Stream Chaining prefetchers and G/DC. 256KB L2 cache.

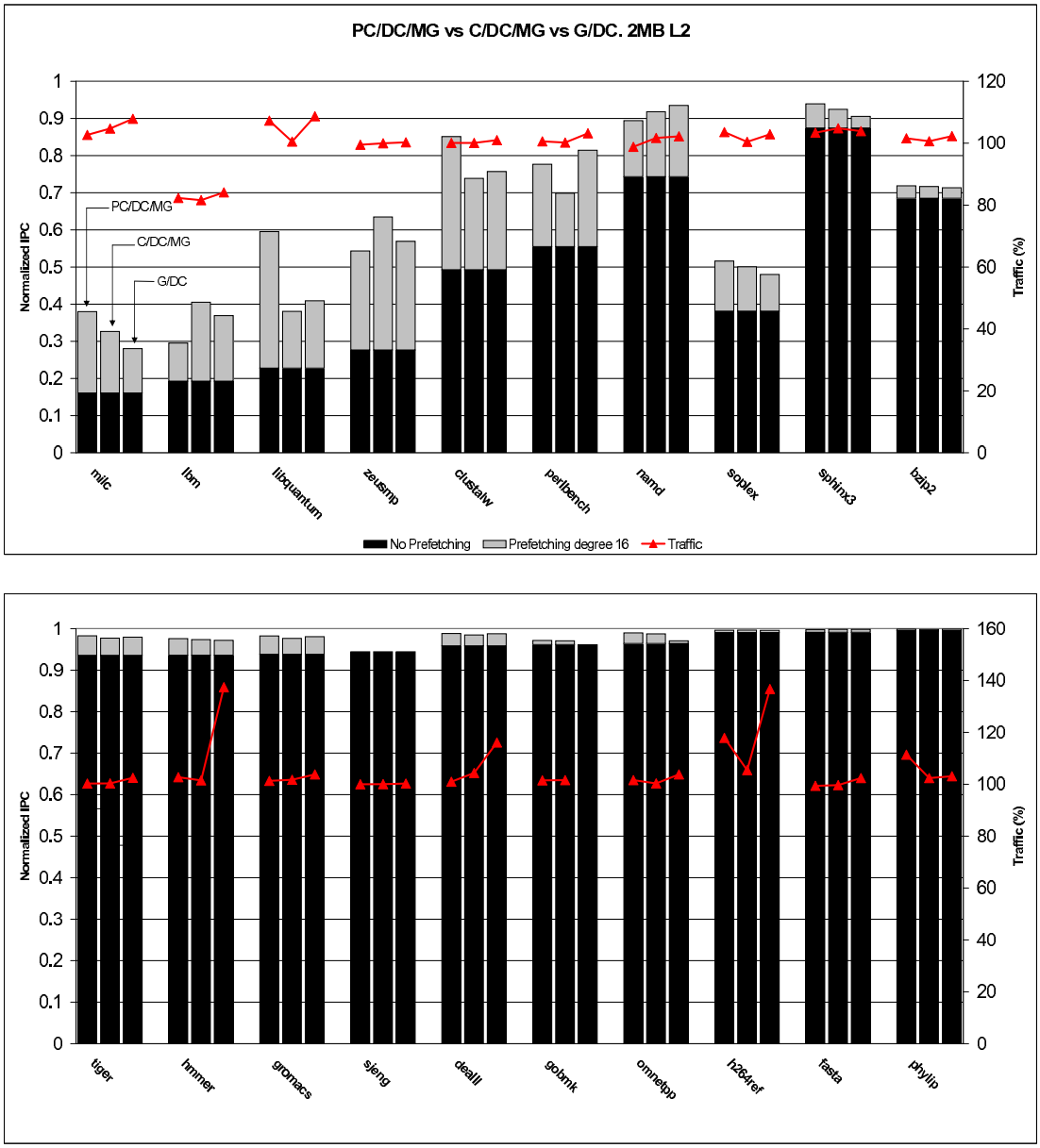


Figure 4.25: Comparison of Stream Chaining prefetchers and G/DC. 2MB L2 cache.

Chapter 5

Resizable Prefetch Heaps

5.1 Introduction

This chapter introduces the concept of *Resizable Prefetch Heaps* (RPH), the second main contribution of this dissertation. The RPH is a novel organization of the Prefetching Request Queue (PRQ, Section 2.6.2) that allows prefetch throttling (Section 2.7) in multi-core systems (Section 2.8) to be performed at the PRQ level. In order to do so, prefetch requests are assigned a priority that reflects their relative value among the pool of pending prefetch requests. Prefetch requests are extracted from the queue according to their priority, effectively turning the PRQ into a logical priority queue. In case of overflow, requests already in the PRQ are only overwritten if the incoming request has greater priority (i.e., they are *judged* to be more valuable). In addition to this, the size of the RPH PRQ is adaptively changed according to the current memory channel utilization.

Prefetch throttling is especially important in multi-core systems, where each core's prefetcher competes with the other prefetchers for memory bandwidth. Because of this, we assume in this chapter a multi-core architecture with prefetching as described in Section 2.8.

The rest of this chapter is organized as follows: Section 5.2 characterizes prefetch throttling as a producer-consumer problem; Section 5.3 describes *HPAC*, a state-of-the-art prefetch throttling mechanism on which we base part of the behavior of the RPH; finally Section 5.4 introduces the *RPH* PRQ and describes in detail its operation.

5.2 Prefetch Throttling as Producer-Consumer Problem

The problem of prefetch throttling can be viewed as a generalization of the well known producer-consumer problem. Prefetch requests are generated by one or more producers, the prefetch engines. In a simple single-core system there will be just one producer, while on more sophisticated systems and in multi-core systems there will be a group of producers. On the other hand, typically we have just one consumer: the memory controller, although more sophisticated systems might have more than one memory controller and, thus, more than one consumer. The memory controller is in charge of consuming the prefetch requests generated by the producers as soon as conditions allow. Normally memory controllers will wait for the memory bus to be idle before dispatching any prefetch requests, as these are usually given lower priority than normal memory requests. A temporary data store called the Prefetch Request Queue (PRQ, Section 2.6.2) sits between the consumer and the producers, in order to allow queueing of pending prefetch requests.

An important difference between this setup and the traditional producer-consumer problem is that not all the data items produced need to reach the consumer. In fact, the main objective of a prefetch throttling system is to ensure that only the prefetch requests that are thought to be more desirable do reach the memory controller. Note that in the most general formulation of this problem, the “desirability” of a prefetch request is re-assessed at every time step and is measured considering all the currently pending prefetch requests. Obviously, such ideal control is not practical and compromises have to be made.

Viewed in this light, it is clear that current prefetch throttling mechanisms (Section 2.7) work only on the producer side of the problem. Be it by filtering out prefetch requests or reducing the prefetch degrees, these mechanisms operate before the prefetch has reached the PRQ, at each prefetching source (the producer in our analogy). Once a prefetch request has passed all the filtering and throttling mechanisms, it is enqueued in a simple linear FIFO and consumed in turn. By not tackling the consumer side of this problem, throttling mechanisms ignore two important aspects of the producer-consumer problem: how to consume the prefetch requests in the most effective manner and how to discard elements in case the prefetching queue fills up. As a consequence, most prefetch throttling algorithms err by being too cautious in the type of prefetches that they let through. Since each prefetch request that is inserted in a conventional FIFO-like PRQ may overwrite other possibly more important requests, or be put in

front of more time-critical requests (disturbing their timeliness), current throttling algorithms cut down prefetch requests for which there are not enough guarantees that they will be beneficial. This binary good/no good approach leads to lost opportunities in prefetching.

In this chapter, we describe a way to throttle prefetchers that works both on the producer and consumer side of the problem. With RPH throttling, each prefetch is graded in such a way that obviously good prefetches are always dispatched, possibly good prefetches are dispatched if there is enough resources for them, and bad prefetch requests (i.e., those that clearly degrade performance) are filtered out. We also provide a mechanism that manages overflow in the PRQ and discards the prefetch requests judged possibly less beneficial. Lastly, we include a global throttling down mechanism that reduces the number of prefetch requests at times where the memory bus is saturated, regardless of other metrics.

Before delving into the details of operation of the RPH, we first set the context of prefetch throttling in the next section, describing a state-of-the-art prefetch throttling mechanism known as HPAC.

5.3 Case Study: The Hierarchical Prefetch Aggressiveness Control

Traditionally prefetch throttling algorithms have used metrics local to the prefetcher being throttled in order to make decisions about its aggressiveness. This local-metrics-only approach may be suitable for single-core architectures, but in a multi-core system it ignores important issues such as inter-core prefetching pollution and bandwidth sharing. Only very recently there have been proposals to take global metrics into account. In this chapter we describe HPAC [24] one of such prefetch throttling algorithms that uses global and local metrics to make throttling decisions. When we introduce the RPH PRQ in Section 5.4 we use some of the basic principles behind HPAC to build our prefetch throttling strategy.

The HPAC throttling mechanism is tiered in two decision layers. In the first layer, a set of rules watch for harmful interactions of prefetchers from different cores. In such case, the interfering prefetcher(s) is(are) throttled down to avoid global performance degradation. On the other hand, if no such interferences are found, control is passed to a local decision layer which runs a set of rules based on local metrics (accuracy, local

Decision layer	ACC_i	BWC_i	POL_i	$BWOC_i$	Action
Global	low	-	low	high	Global throttle down
	low	low	high	low	
	low	high	high	low	
	low	-	high	high	
	high	high	high	high	
Local	low	high	low	low	Local throttle down decision
	high	high	high	low	
	high	low	high	high	
	Any other case				Local decision

Table 5.1: HPAC decision rules

pollution, etc.) and regulates the aggressiveness of each prefetcher accordingly.

Table 5.1 summarizes the rules used in HPAC throttling. This set of rules is evaluated periodically, once per core in the system. At evaluation time, four basic metrics are used: ACC_i, the accuracy of the prefetcher i; BWC_i, the bandwidth consumed by core i; POL_i, a measure of the pollution caused by prefetcher i to other cores $j \neq i$ [1] and $BWOC_i$, the sum of bandwidth needed by all cores $j \neq i$.

As it can be seen in table 5.1, the global and local decision layers can spawn three types of actions. The first group of actions, activated by the global decision layer, is run when HPAC detects severe inter-core interference from one prefetcher. HPAC then reduces the aggressiveness of that prefetcher in order to alleviate the situation. The local decision layer can run two type of actions. The first one corresponds to a borderline interference scenario. In this case the local decision layer role is to avoid a given prefetcher to transition to a severe interference scenario. For this, the local decision layer rule set is run to possibly throttle down (based on local metrics) but never throttle up the prefetcher. Lastly, when there is no detectable inter-core interference, the local decision rule set is run to adjust the prefetcher aggressiveness based on metrics local to that prefetcher. HPAC uses *Feedback Directed Prefetching* [23] as the local prefetch control heuristics.

The throttle down and up actions lower and increase respectively the aggressiveness of a prefetcher. This is done increasing or decreasing the prefetch degree, and, in the case of the *stream prefetcher* (Section 2.5.2.1), the prefetch distance. For practical reasons, the values the prefetch degree can be set to are usually limited to a reduced set

[1] This refers to the number of cache lines belonging to other cores that have been evicted this core prefetches and later referenced again.

of possible values (for example, uniformly sampling the configuration space from the least aggressive setting to the most). Therefore, in this context, increasing (or decreasing) the prefetch degree implies setting it to the immediately next higher (or lower) value from the set of degrees. Since the exact composition of the set of prefetching degrees is implementation dependent, throughout this Chapter we will refer only to the throttle up or down operations and not to the particular degree the prefetch is adjusted to.

5.4 Resizable Prefetch Heaps

In this section we present our proposal for throttling prefetches in a multi-core. The key idea of the proposal is to perform *global* throttling, which is achieved through a novel organization for the prefetch queue: the *Resizable Prefetch Heap (RPH)*. The RPH employs two new techniques for prefetch throttling: prioritization of prefetch requests according to prefetch metrics and resizing of the RPH according to available bandwidth.

5.4.1 Introduction

The key novel infrastructure proposed in this dissertation to achieve global throttling is the Resizable Prefetch Heap (RPH). The RPH is designed as a drop-in replacement for ordinary FIFO prefetch queues. As its name suggests, the RPH is based on the binary heap data structure, explained in Section 5.4.1.1. The RPH works as a priority queue, where each prefetch request has an associated priority. At de-queue time (performed by the memory controller), the prefetch request with highest priority is selected and extracted from the RPH.

Another property of the RPH is the ability to resize itself according to the current memory channel utilization. At times where the memory channel is saturated by demand requests from the processor cores, the RPH shrinks in size, reducing the number of prefetch requests sent to memory and acting as a global throttling mechanism. The RPH is implemented logically as a linear array of queue elements (Sections 5.4.1.1 and 5.4.1.2). Therefore resizing the RPH can be done seamlessly by storing in a register the current size of the heap, and ignoring elements in the array past this limit.

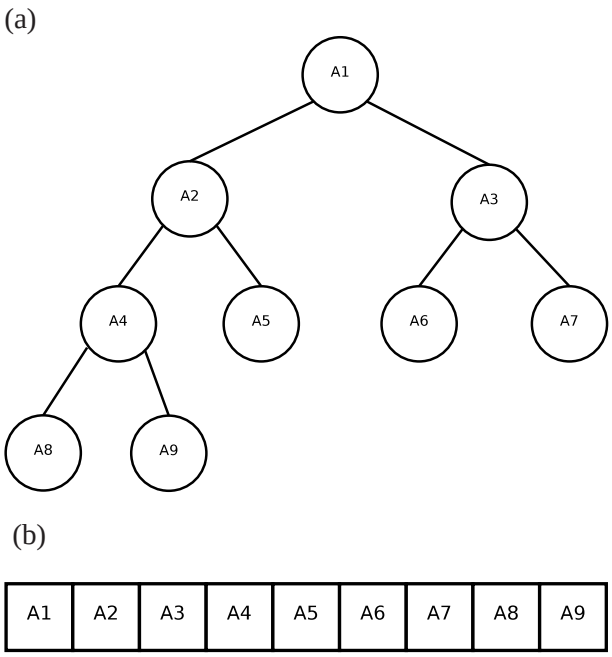


Figure 5.1: Logical view of a binary heap: (a) as a tree; and (b) as an array.

5.4.1.1 Binary Heaps and Priority Queues

Binary heaps are well known data structures [1]. They are array objects that can be viewed as a nearly complete binary tree (Figure 5.1a), with possibly the lowest level incomplete. Each node in the tree corresponds to one element in the array. This mapping is done in the following way: the root of the tree is the first element of the array, and for each element $A[i]$ in the array its two children in the tree are the array nodes $A[2*i]$ and $A[2*i+1]$ (Figure 5.1b).

An important property of binary heaps is that, viewed in tree form, each node is equal to or bigger than any of its children [2] (for some definition of equal to or bigger than previously established). This property allows heaps to be used as an efficient implementation of priority queues. Queueing and de-queueing from a heap-based priority queue can be done efficiently in $O(log_2 n)$ operations, where n is the size of the queue. The basic method of insertion in a heap consists in inserting the element at the bottom of the heap (the lowest level of the tree) and successively swapping this element with its parent until the heap property is restored. Extraction from a heap-based priority queue is done by removing the root element from the tree and putting in its place the bottom-right (last element in the array) element of the tree. Then the new root ele-

[2] More precisely this defines max-heaps, where the biggest element in the heap is the root of the tree. Conversely, in min-heaps the root of the tree contains the smallest value in the heap.

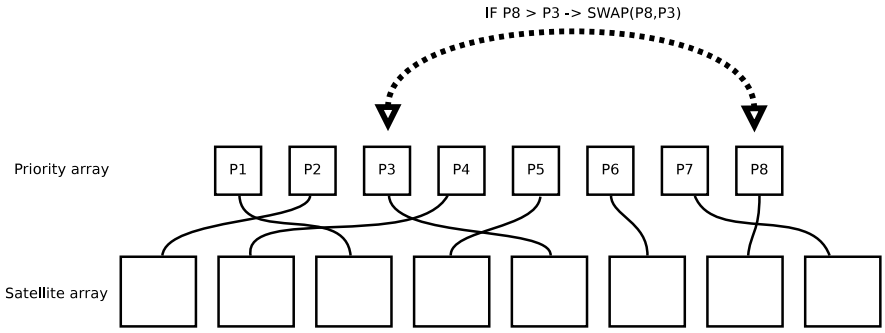


Figure 5.2: Heap array split into priority and satellite data arrays.

ment is again successively swapped with one of its children until the heap property is restored.

5.4.1.2 Hardware Implementation of Binary Heaps

Binary heaps are suited for hardware implementation because of their array representation. In hardware, a binary queue with up to a few hundred items can be directly implemented with just a normal array store and combinatorial circuits [37]. Each item in the array store contains two elements: a priority field and a data field. The priority field is used to organize each item within the queue, while the data field contains the satellite information associated with each element. Since binary heap operations require swapping elements in the array, this hardware organization might not be convenient if the data field is relatively large. This can be solved by using an additional satellite data array that holds the data. In this case the heap array only holds the priority and a pointer to the satellite data array, as shown in Figure 5.4.1.2. Compared to a circular FIFO queue, the hardware overhead of binary heaps is minimal.

One possible issue with the use of binary heaps in hardware is that insertion and deletion from them are no longer constant-time operations, but logarithmic. We do not anticipate this to be a problem. In our proposed use, we will model a queue with a maximum of 256 entries. This would mean that insertion or deletion in the queue would take a maximum of $log(256) = 8$ operations. Note that, although each operation requires one comparison and one swap step, these do not necessarily need to be performed sequentially. An optimized hardware implementation can take advantage of the fact that, when inserting into the heap, all comparisons performed, except the last one, have the same sign. Therefore such optimized implementation can perform the swap and comparison steps in parallel in one cycle, with the caveat that when

the comparison changes sign, the last swap was unnecessary and needs to be undone. Thus, inserting into the queue can be done in at most $log_2(256)+1=9$ cycles, well within the timing constraints for current L2 caches. Furthermore, with this optimized implementation, insertion operations can be pipelined: in case of having two or more consecutive insertions into the heap, each insertion can start just two cycles after the preceding one [3]. Lastly, if strict constant-time insertion and deletion are needed, this can be achieved by placing one set of comparators for each element in the array store and implementing a sorting network [38], which would perform all the comparison in parallel. As with most hardware implementations, there is a performance/cost trade-off that must be taken into account.

Even if a non-optimized hardware implementation is used, note that binary heaps are in fact almost-full binary trees. As such, approximately 50% of the elements of the heap will be leaves of the tree (a full binary tree with 2^n-1 elements has 2^{n-1} leaf nodes), and 75% of all elements are located within the first two bottom levels of the tree. Therefore, when inserting a new element into the heap, it is likely that it will move only a few levels upwards the heap, requiring on average many fewer compare-swap operations than the worst-case $O(logn)$.

Extraction in an unoptimized binary heap is likely to carry a longer latency than insertion. This should not be a problem because extraction has more lenient timing constraints, since the requests are being dispatched to memory by the memory controller, which will give priority to regular cache misses over prefetch requests. Even so, there are some extraction optimizations that can be performed. First, extraction can be divided in two separate and independent tasks: 1) extraction of the value (the root of the tree); and 2) heap rebuilding. The first task can be completed immediately, since the root of the tree is the first element of the heap array. Therefore the extraction procedure can return a prefetch request to the memory controller in just one cycle, with the caveat that the next extraction can not happen until the heap is rebuilt, which can take up to 8 cycles in our case [4]. Moreover, the top two levels of the tree contain only three elements. An optimized implementation could simply perform a parallel comparison-swap between them in one step instead of two. This could even be extended to the next level at the cost of more complicated comparison hardware.

[3] A space of two cycles is needed to allow for the last swap-undo step corresponding to the preceding operation

[4] Note that extraction and insertion can run in parallel, so there is no need to lock the queue while the extraction heap rebuilding is happening.

Memory utilization	CSR
$\leq 30\%$	Hardware Size (HS)
$\leq 50\%$	$0.5 \times HS$
$\leq 70\%$	$0.3 \times HS$
$\geq 70\%$	1

Table 5.2: Memory utilization scale used to resize the RPH.

5.4.2 Prefetch Throttling with RPHs

RPHs are basically a hardware implementation of a binary-heap max-priority queue with adaptive resizing. Extraction from the queue gives back the element with the highest priority in it. Insertion in a full queue only modifies it if the priority of the item to be inserted is greater than the lowest priority stored in the queue. Therefore, the behavior of an RPH is governed by two main design decisions: how to assign a priority to each prefetch request and how to perform the adaptive resizing.

5.4.2.1 Adaptive Resizing

Resizing of the RPH is done by storing in a register the current size of the heap. We call this the Current Size Register (CSR). The contents of the CSR can be equal to or less than the maximum storage size of the queue (Hardware Size, HS). Insert and extract operations take into account CSR and do not consider array positions bigger than it. We recalculate the CSR in interval windows of a million cycles. At the end of each interval we calculate the utilization of the memory channel and change the CSR according to a scale (Table 5.2). In order to calculate the memory channel utilization, we count how many misses we register during the window (not including prefetch requests) and normalize it to the maximum possible number of misses during that period.

5.4.2.2 Priority Assignment

Assigning priorities to prefetch requests defines the main throttling behavior of the RPH. For this, we base our prefetch control heuristics on those used by the state-of-the-art throttling mechanism HPAC [24]. Moreover, we use the same set of rules as HPAC (Table 5.1). However, crucially, we construct a system of prefetch priorities that allow any prefetch request to be potentially added to our RPH PRQ. For this, we assign each prefetch request a priority based on the following formula:

$$\text{Prio}(\text{W}_n, P_{ctr}, Acc, Allow, Off set) = \begin{cases} (32 - W_n) - P_{ctr} - \text{Offset} + \text{AccBonus} & \text{if Allow} == true \\ 0 & \text{otherwise} \end{cases} \quad (5.1)$$

$$\text{AccBonus} = \begin{cases} 5 & \text{if Acc} \geq 0.9 \\ 3 & \text{if Acc} \geq 0.6 \\ 0 & \text{otherwise} \end{cases} \quad (5.2)$$

For any given prefetch request, we calculate its priority based on five pieces of information: W_n, the prefetch wave number; P_{ctr}, the prefetch activation counter; *Acc*, the accuracy of the prefetcher issuing the request; *Allow*, a boolean variable that is fed back from the throttling heuristics; and *Off*, an optional offset explained bellow.

Clearly not all prefetch requests should be enqueued into the RPH PRQ, no matter how low their priority. More concretely, those prefetches deemed harmful by the global control layer should be removed. This is what the $Allow$ boolean variable implements. This variable is fed back from the global control layer of our throttling heuristics and is set to zero when one prefetch request is deemed harmful or causing severe interference.

We calculate prefetch request priorities using Equations 5.1 and 5.2. In the case of $Allow = 0$, the prefetch request is given priority zero, which is treated as a special case meaning that it should be ignored and not inserted in the queue. The prefetch wave number is simply the relative ordering of a prefetch request within a single prefetcher activation (i.e., the set of prefetch requests issued by one prefetcher after a given miss). For each prefetcher activation, the first prefetch issued would have a wave number 1, the next one would have a wave number 2, and so on up to the prefetch degree. The prefetch activation counter (P_{ctr}) stores the total count of prefetcher activations (in any core) in the system. Lastly the accuracy of the prefetcher is stored normalized to 1.

Term by term formula analysis: Below we describe the rationale behind each term in the priority assignment formula.

- $(32 - \text{W}_n) \rightarrow$ *Wave ordering:* In almost every prefetcher the accuracy of the prefetch requests diminishes as the wave number increases, since we are predicting further and further into the future. Therefore, we subtract the wave number from 32 (the maximum prefetch degree we consider in our evaluation) so that prefetch requests with lower wave numbers have higher priority than requests with higher wave numbers.

- $-P_{ctr} \rightarrow$ *Time ordering:* In order to avoid data stalling issues, we relate each

prefetch request to the prefetch activation number. By subtracting this number from the priority we achieve a gradual prioritization of older requests in the queue: since this is an ever increasing counter, older requests will progressively have more priority than newer requests[5].

- $-$Offset $\rightarrow$ *Prefetch classing:* We permit certain classes of prefetches to have an overall *lower* priority than the rest of requests. We do so to distinguish between prefetches that the throttling heuristics judge surely beneficial to other prefetches which are only *possibly beneficial*. For the first class we use Offset $= 0$ while for the second class we use a suitable large value (Offset $= 100$ in our experiments).

- AccBonus $\rightarrow$ *Accuracy bonus:* Based on experimental results, we have determined that promoting highly accurate prefetches improves the overall prefetching performance of the system. For this reason we use AccBonus, which is a variable that contains a priority boost of 5, 3 or 0 depending on the accuracy of the given prefetcher (values determined empirically).

Overall formula behavior: The main objective of the priority assignment formula is to interleave prefetching requests from several cores in such a way that, in case of competition between prefetchers, no single prefetcher can completely overtake all the space in the PRQ. Within a single prefetch activation we enforce the time ordering in which the prefetch requests were generated. When two or more prefetcher activations insert several requests into the queue, our priority assignment maintains a general interleaving of requests while gradually promoting (and flushing) older requests over new ones. Prefetchers that show high accuracy are slightly promoted over the rest, but the effect of this promotion is limited (the maximum priority boost of 5 means a request can "jump" up to five places in the queue) and the general interleaving is still enforced. Lastly we allow a class of "optional" prefetch requests that have lower overall priority than primary requests but whose dispatch is still interleaved (within their class) in the same way.

5.4.2.3 Integration with Throttling Heuristics

So far we have discussed how an individual prefetch is assigned a priority according to the priority assignment formula, which in turn depends on five signals or variables

[5] In order to prevent overflow, this counter should be reset at context switch time.

Decision layer	HPAC action	RPH action
Global	Global throttle down	Reduce D_{limit} and possibly D by one step
Local	Local throttle down decision	Local reduce D decision
	Local decision	Set D and D_{limit}

Table 5.3: RPH throttling actions.

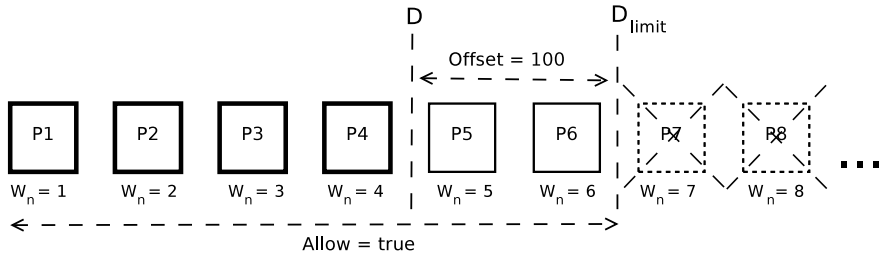


Figure 5.3: Role of D and D_{limit} registers. W_n denotes the prefetch request wave number.

(W_n, P_{ctr}, *Acc*, *Allow* and *Offset*). In this section we describe how to integrate the HPAC throttling heuristics with the RPH so they drive these signals.

In a conventional system throttled by HPAC heuristics, the global decision layer has the power to limit the aggressiveness of a given prefetch in case it is causing excessive inter-core interference or pollution. To do this, it throttles down such prefetcher with a clear cut on the degree. Similarly, we do not allow clearly harmful prefetches in the RPH. For this we use a register called D_{limit}, which stores the maximum wave number for which the *Allow* signal is set; any other prefetch requests after D_{limit} are effectively discarded, as per the priority assignment formula (Equation 5.1). In this aspect, both RPH throttling and conventional HPAC throttling work in a similar way. Thus, unlike HPAC, we also allow for some deeper prefetches to conditionally make it into the queue, depending on the priority values of the other requests already in the PRQ. This allows for a more graceful global control. Additionally we use another register called D: any prefetch whose wave number is greater than D activates the *Offset* signal, which marks this prefetch request as "optional" and gives it a lower priority. In summary, comparing the wave number with D and D_{limit} permits calculating the *Offset* and *Allow* signals, which in companion with the P_{ctr} counter and the accuracy metric of the prefetcher, are sent to the priority calculation hardware. Figure 5.3 illustrates the role of D and D_{limit} for setting the signals used in priority calculation.

We modify the throttling heuristics to operate on D and D_{limit} instead of actuating directly over the prefetcher (Figure 5.3). If the global decision layer rules are triggered, the RPH reduces the value of D_{limit} (by choosing the immediate next lower value from the prefetch degree set). If, as a result of this, D is greater than D_{limit}, D is set back to the same value as D_{limit}. Once in the local decision layer, there are two alternatives. In the borderline scenario, where the local decision rules are run to possibly throttle down a prefetcher, we reduce instead the value of D_{limit} (again, by choosing the immediate next lower value from the degree set). Similarly to the global layer case, we reset D to D_{limit} if as a result of this action it would remain bigger than it. In any other case, the local layer rules set the value of D. D_{limit} is in this case set to the immediate next higher degree in the prefetch degree set.

Chapter 6

Evaluation of Resizable Prefetch Heaps

6.1 Simulation Setup

As with the evaluation of Stream Chaining prefetchers (Chapter 4), we use SESC[30] for all our simulations. We simulate a modern multi-core system with aggressive out-of-order cores and an unified L2 cache. We focus our experiments on a 8-core system with limited bandwidth, which we consider representative of future multi-core designs, where the number of cores is expected to grow more rapidly than the memory technology bandwidth. Table 6.1 summarizes the architectural parameters of the simulated system.

We skip the data loading phase of each benchmark. For the PARSEC benchmarks (Section 6.2), this is done skipping until the `parsec_roi_end` segment is found. For all the other benchmarks, this point was found inspecting the source code. We then simulate in detail and collect statistics for the next 1 billion instructions/core[1].

6.2 Benchmarks

We use a mixture of workloads from the PARSEC and ALPBench benchmarks suites, as well as a popular stand-alone parallel compression program (pbzip2) and a purely scientific generic workload from the NAS parallel benchmarks (Conjugate Gradient calculation, *CG*) . We chose these workloads for two reasons: they offer a good sample

[1] In the parallel benchmarks we actually stop simulation at the next synchronization point after 1 billion instructions/core.

Parameter	Value
Core Frequency	5GHz
Fetch/Issue/Retire Width	6/ 4/ 4
I-Window/ROB	80/ 152
Branch Predictor	64kbit 2BcgSkew
BTB/RAS	2k entries 2-way/ 32 entries
Minimum misprediction	20 cycles
Ld/St queue	8
L1 ICache	2-way, 64B lines, 2 cycles
L1 DCache	64KB, 4-way, 64B lines, 2 cycles
L1 MSHR's	4
L1-L2 bus	64bits
Prefetch Algorithm	C/DC (64KB CZones)
Prefetch degree	32
IT	512 entries, 1 cycle
GHB	512 entries, 5+1*hop cycles Access is mutually exclusive and preemptive: new requests drop in-execution requests

Parameter	Value
Number of cores	8
L2 cache	4MB. 64B lines
L2 associativity	16
L2 MSHR's	32
Prefetch Request Queue	FIFO, RPH
PRQ size	256 slots
Prefetch throttling	HPAC, RPH
Memory latency	400 cycles
Memory bandiwth	12.5 Gbps

Table 6.1: Architectural parameters: Per core (left) and system-wide (right).

of current engineering, scientific and media processing applications in use nowadays and they are available in parallel and sequential versions.

Because of limitations in the simulator used, some benchmarks could not be ported to our simulation environment. Appendix A gives a detailed description of all the benchmarks evaluated, as well as the reference input data sets used.

We consider two scenarios: parallel and multi-programmed execution. In the parallel scenario we run the parallelized version of each workload with 8 threads. Most workloads were parallelized with the explicit use of a threading library, with the exception of *bodytrack*, *CG* and *freqmine*, which were parallelized using OpenMP directives.

To simulate the multi-programmed scenario we use 10 random groupings of 8 programs from the benchmark suites described above. Each program runs in sequential mode, and therefore there is no communication or synchronization of any kind between cores. Table 6.2 lists the programs we run on each multi-programmed workload.

6.3 Prefetch Mechanism and Throttling Strategies

Current research in throttling and filtering algorithms has favored the Stream Prefetcher [23, 24, 27] in their evaluations, as it is the prefetching algorithm most frequently implemented in current hardware. This is due to its simplicity and relative good performance. However, stream prefetchers cannot prefetch complex memory patterns and are prone to high cache pollution when they misspredict. On the other hand, research in

Workload	Benchmarks
MP8_1	CG, blackscholes, ferret, fluidanimate, freqmine, streamcluster, tachyon, x264
MP8_2	CG, MPGdec, MPGenc, blackscholes, dedup, ferret, fluidanimate, streamcluster
MP8_3	blackscholes, dedup, ferret, fluidanimate, freqmine, streamcluster, tachyon, x264
MP8_4	MPGdec, blackscholes, dedup, ferret, freqmine, streamcluster, tachyon, x264
MP8_5	blackscholes, canneal, dedup, ferret, fluidanimate, pbzip2, streamcluster, tachyon
MP8_6	MPGdec, MPGenc, canneal, ferret, freqmine, pbzip2, swaptions, x264
MP8_7	CG, MPGdec, MPGenc, fluidanimate, pbzip2, streamcluster, tachyon, x264
MP8_8	CG, MPGenc, canneal, fluidanimate, freqmine, pbzip2, swaptions, tachyon
MP8_9	MPGenc, blackscholes, dedup, ferret, fluidanimate, pbzip2, swaptions, tachyon
MP8_10	MPGenc, canneal, dedup, fluidanimate, freqmine, streamcluster, swaptions, x264

Table 6.2: Multi-programmed workloads.

prefetching algorithms has offered a variety of more complex methods. As the transistor count increases in each new processor generation, we expect that future processors will implement more sophisticated (and more accurate) prefetching algorithms. For this reason we use the C/DC [11] prefetcher implemented using the Global History Buffer [10] for our evaluation. C/DC is a modern accurate prefetcher that is less likely to pollute the cache than the stream prefetcher.

We implement two throttling strategies for our experiments. In our baseline throttling configuration we simulate the *Hierarchical Prefetcher Aggressiveness Control* (HPAC, [24]) with a conventional FIFO-based PRQ. We compare this to an RPH configuration with prefetch throttling enabled in the PRQ. Both strategies are described in detail in Chapter 5. In both cases the size of the PRQ is 256. In order to modify prefetcher aggressiveness, HPAC can vary the prefetch degree of each prefetcher to one of the following values: $\{1, 4, 8, 16, 32\}$. RPH throttling is run with the prefetchers set to the most aggressive setting (degree 32), but as explained in Section 5.4 the actual number of prefetch requests accepted for inclusion in the RPH PRQ depends on their priority.

For implementing the RPH, we simulate a hardware binary heap with separate priority/satellite arrays, pipelined insertions, and split extraction procedure, as explained in Section 5.4.1.2. Additionally, we collect statistics on the number of comparison-swap steps needed to insert and extract elements from the queue without the insertion and extraction optimizations.

6.4 Metrics

We evaluate our results against three main metrics: performance increase, memory bus traffic increase and prefetch fairness.

6.4.1 Performance

For multi-programmed workloads we measure performance as the *Harmonic Speedup of IPCs* against a configuration with no prefetching (Equation 6.1).

$$\textbf{HarmonicSpeedup} = \frac{N}{\sum_{i=1}^{N} \frac{IPC_i^{NoPref}}{IPC_i^{Pref}}} \tag{6.1}$$

For parallel applications we measure the execution time of the application between two synchronization points and calculate the speedup against a configuration with no prefetching. We also calculate the benefits of a baseline throttling configuration (HPAC with a conventional FIFO PRQ) compared to a medium-aggressive fixed-degree prefetching configuration. We then calculate the percentage increase of this metric when we use RPH.

6.4.2 Traffic

We measure the total memory bus traffic in the several prefetching strategies we simulate. We then compare the increase in traffic against a configuration with prefetching disabled.

6.4.3 Prefetch Fairness

With this metric we aim to quantify how well a prefetch delivery mechanism maintains the benefits of prefetching for a certain processor core in the presence of other competing prefetches for other cores. In other words, we try to measure the variance in prefetching performance improvement for a given application in the context of other applications running on other cores (with other prefetching engines) in a multi-core system. Note that this is different from the metrics proposed in [28], which measure the variance in execution time of a thread in the presence or absence of other threads that share the same cache. We discount the effect of cache sharing fairness in our metric by measuring variance from the same multi-programmed configuration with and without prefetching enabled.

To do this, we evaluate the performance (measured in number of cycles needed to execute a certain number of instructions) of a sequential application running with no prefetching in several multi-programmed configurations $C_1, C_2, C_3, ...C_n$. We denote each timing measurement $T^{NP}_{C_1}, T^{NP}_{C_2}, T^{NP}_{C_3}, ..., T^{NP}_{C_n}$. Next, we measure the performance of the same workload, this time with prefetching enabled, across the same multi-programmed configurations. We denote these timing measurements $T^{P}_{C_1}, T^{P}_{C_2}, T^{P}_{C_3}, ..., T^{P}_{C_n}$. We define the Prefetching Fairness of that sequential application as the minus standard deviation of the set of pairwise differences between the timing measurements with and without prefetching. That is:

$$\Delta_i = T^{NP}_{C_i} - T^{P}_{C_i} \qquad \forall i \in [1..n] \tag{6.2}$$

$$\textbf{PrefetchFairness} = -stdev(\Delta_1, \Delta_2, \Delta_3, ..., \Delta_n) \tag{6.3}$$

Note that the measurements used in Equation 6.2 are for a specific application within a multi-programmed workload. By measuring pairwise differences between the same multi-programmed workload we aim to single-out the only factor different between them: prefetching. Furthermore, by measuring the performance contribution of prefetching for the same application across several multi-programmed workloads we aim to isolate the effects of the prefetch delivery mechanism.

As an example to illustrate how prefetch fairness is calculated, Table 6.3 shows how to calculate this metric for the benchmark *fluidanimate* using samples from 5 different multi-programmed configurations. For each multi-programmed configuration, we obtain the number of cycles in which *fluidanimate* ran without prefetching (second column), the number of cycles for HPAC throttling (third column), the number of cycles for RPH throttling (fourth column) and the pairwise differences between the latter two and the non-prefetching configuration (fifth and sixth columns). We perform these measurements for 5 multi-programmed configurations, and take the minus standard deviation of the pairwise differences. Note that we make the standard deviation negative for convenience, since it is more intuitive to say that the greater the number, the greater the fairness is. Finally, we observe from the results that the prefetch fairness of RPH throttling is bigger (i.e., less negative) than the prefetch fairness with HPAC throttling.

Therefore, a prefetching mechanism that exhibits homogeneous speedups for a given application regardless of the multi-programmed execution context will have a higher (i.e., less negative) Prefetch Fairness metric than another prefetching mechanism in which the performance gains are more context dependent and thus show higher

i	T_i^{NP}	T_i^{HPAC}	T_i^{RPH}	$\Delta_i^{HPAC} = T_i^{NP} - T_i^{HPAC}$	$\Delta_i^{RPH} = T_i^{NP} - T_i^{RPH}$
1	836264098	759717270	756869338	76546828	79394760
2	836264098	760077974	756065441	76186124	80198657
3	841791881	784396068	772753219	57395813	69038662
4	823475874	750566810	747058237	72909064	76417637
5	817074014	753770645	748542295	63303369	68531719
$-stdev(\Delta_1, \Delta_2, \Delta_3, \Delta_4, \Delta_5)$				-8523860	-5597396

Table 6.3: Example Prefetch Fairness calculation.

variance. Note that although the effects of cache pollution and interference from other cores contribute to make the measurements Δ_n in equation 6.2 different for each multi-programmed configuration, they remain constant across two system configurations in which we only vary the prefetch delivery mechanism.

6.5 Benchmark Characterization

We start our study with a characterization of the parallel benchmarks and multi-programmed workloads used throughout this evaluation. In Section 6.5.1 we analyze the sensitivity of benchmarks and workloads to the L2 cache. This is a good indicator to distinguish which benchmarks will likely benefit more from prefetching. We expand this analysis in Section 6.5.2, where we show the Hit Rate of the L2, as well as its usage, measured in number of accesses per million cycles.

6.5.1 L2 Cache-Performance Sensitivity

Figures 6.1 and 6.2 show the performance sensitivity of multi-programmed and parallel workloads to the L2 cache. In both cases the performance with an ideal (i.e., 100% hit rate) L2 cache is compared against the performance obtained using a normal L2 cache with no prefetching. As explained in Section 6.4.1, we use the harmonic speedup for multi-programmed workloads and a regular speedup for parallel benchmarks. We plot the performance speedup obtained using an ideal L2 cache compared to that of a regular L2 cache, with the objective of determining how tied is the performance of each benchmark to the performance of the L2 cache.

The multi-programmed workloads show an evenly distributed range of behaviors. The workloads *MP8_1*, *MP8_5* and *MP8_8* are the ones that most benefit from an ideal

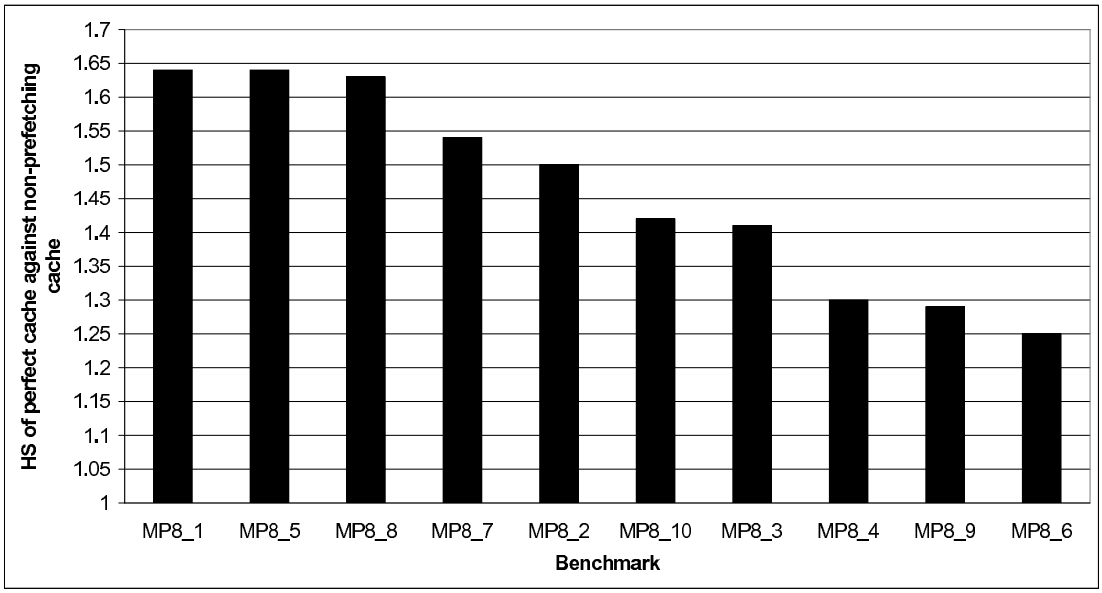


Figure 6.1: Cache sensitivity in multi-programmed workloads.

cache, with harmonic speedups of 1.64 for the first two and 1.63 for the latter. These are the workloads that are most likely to benefit from prefetching, as their performance is highly tied to the performance of the L2 cache. On the other side of the spectrum, the workload *MP8_6* only achieves a 1.25 speedup when using a perfect L2 cache, indicating less dependence to the L2 cache.

The parallel benchmarks show great variance in the performance speedup obtained using an ideal L2 cache. The benchmarks *canneal*, *streamcluster* and *CG* benefit greatly from using an ideal cache, with speedups of 460%, 367% and 349%. On the other hand, the benchmarks *blackscholes*, *bodytrack*, *x264* and *swaptions* are completely L2-oblivious, showing speedups of less than one percent. The rest of the benchmarks show a moderate dependence from the L2, with speedups ranging from 2.2% (*x264ref*) to 13% (*fluidanimate*).

6.5.2 L2 Cache Hit Rates and Usage

Figures 6.3 and 6.5 show the L2 Read Hit Rate (RHR) for multi-programmed workloads and parallel benchmarks respectively. We complement this information with the average number of accesses to the L2 per million cycles executed, shown in Figure 6.4 for multi-programmed workloads and Figure 6.6 for parallel benchmarks.

For the multi-programmed workloads, we can see that in general there is a correlation between the L2 RHR and the performance sensitivity to the L2 cache. The

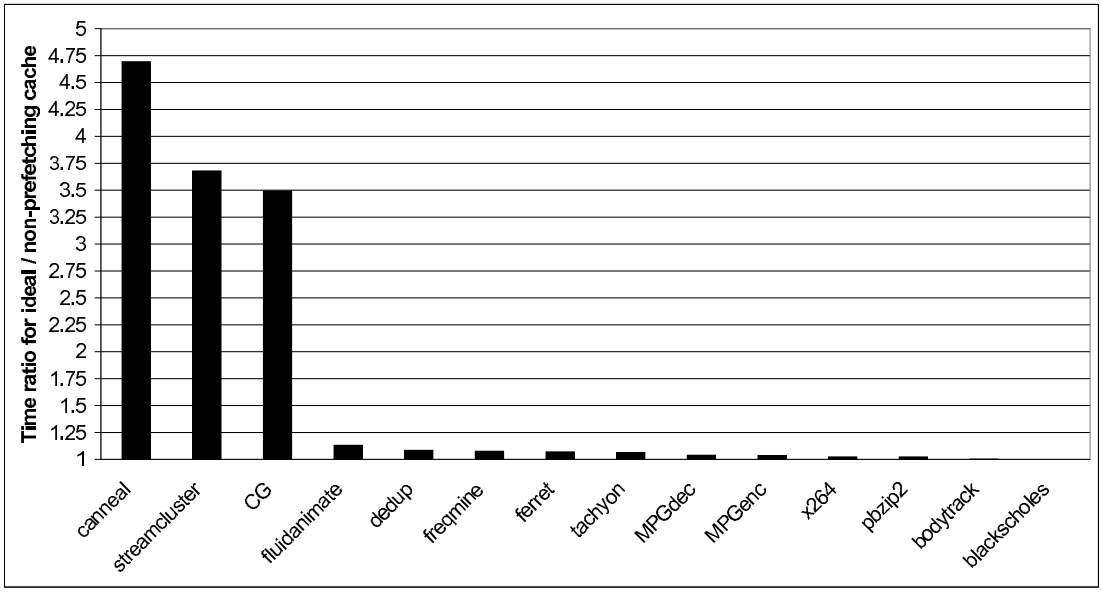


Figure 6.2: Cache sensitivity in parallel benchmarks.

exception to this is the workload *MP8_5*, which shows a significantly higher RHR than other similarly performing benchmarks such as *MP8_1* and *MP8_8*. This benchmark however performs a high number of L2 accesses per million cycles (the second highest for multi-programmed workloads), which explains why its performance is limited by the L2 in spite of a relatively higher RHR.

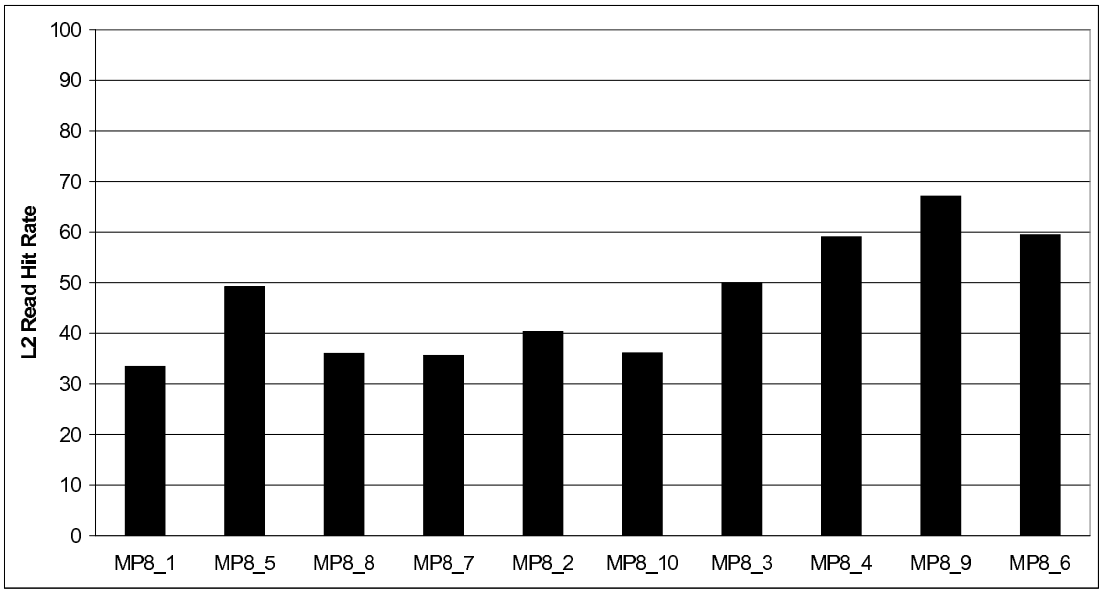


Figure 6.3: L2 Read Hit Rate for multi-programmed workloads.

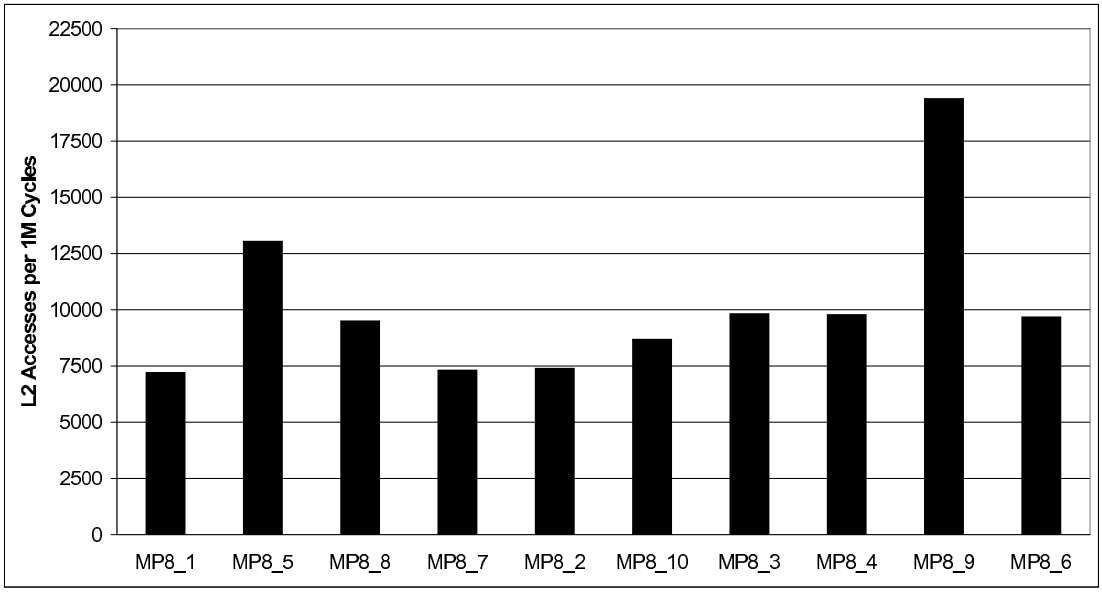


Figure 6.4: Number of accesses to the L2 cache per million cycles for multi-programmed workloads.

The parallel benchmarks show on average a high L2 RHR, with the exception of the benchmarks *canneal*, *streamcluster*, *fluidanimate* and *MPGdec*. The higher L2 RHR in parallel benchmarks compared to multi-programmed workloads can be explained by increased data sharing in the former: most parallel applications share a common programming structure, replicated in each thread. In this case, a significant portion of the data used in computation is either shared between threads (which reduces significantly the memory requirements) or contiguously stored in memory (which facilitates caching). Multi-programmed workloads do not have the advantage of this kind of data sharing.

The low RHRs of the benchmarks *canneal* and *streamcluster* explain the high speedups they obtain when using a perfect L2 cache (Figure 6.1). The benchmark *MPGenc* has as well a low RHR, but this is compensated by its extremely low number of accesses to the L2 per million cycles (the lowest overall), and therefore does not make it L2-bound. Lastly, we note that the benchmark *CG* has a high RHR but overall obtains the third biggest speedup when using a perfect L2 cache. This is due to the very high number of the L2 accesses it shows, the highest among all benchmarks and workloads.

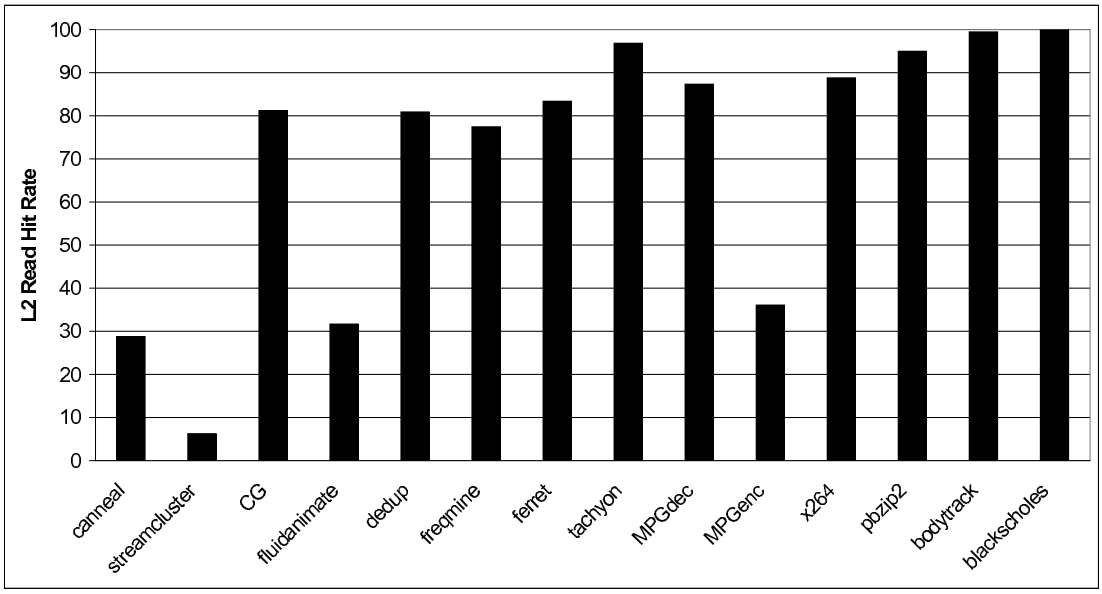


Figure 6.5: L2 Read Hit Rate for parallel benchmarks.

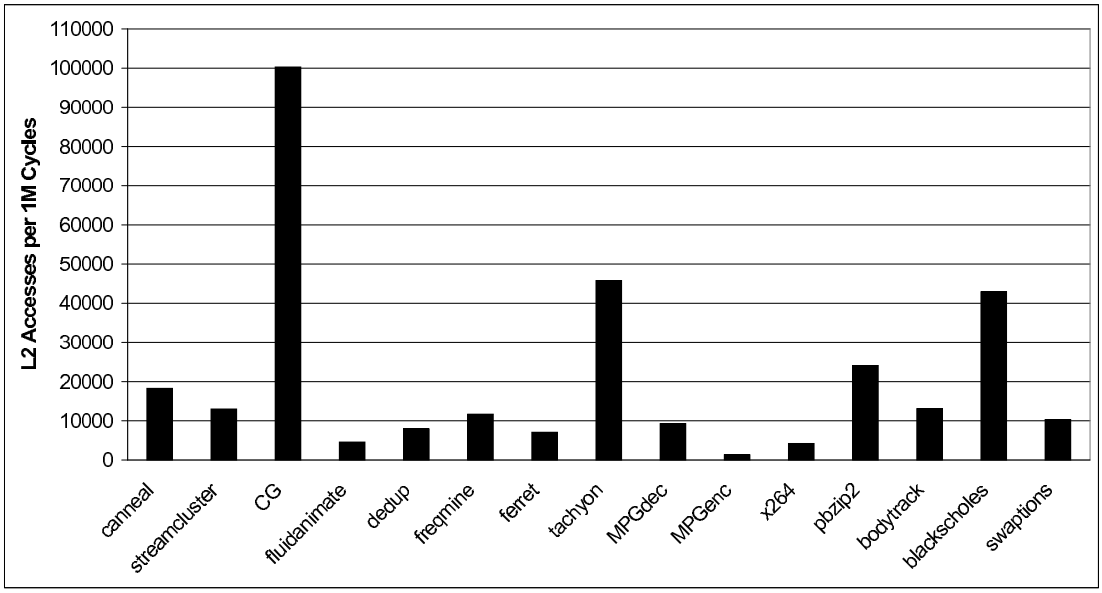


Figure 6.6: Number of accesses to the L2 cache per million cycles for parallel benchmarks.

6.6 Prefetching Performance

Figures 6.7 and 6.8 show the performance of multi-programmed and parallel benchmarks for four prefetching strategies: fixed degree of 24 (medium-aggressive prefetching, *PREF24*), fixed degree of 32 (aggressive prefetching, *PREF32*), HPAC throttling

and RPH throttling.

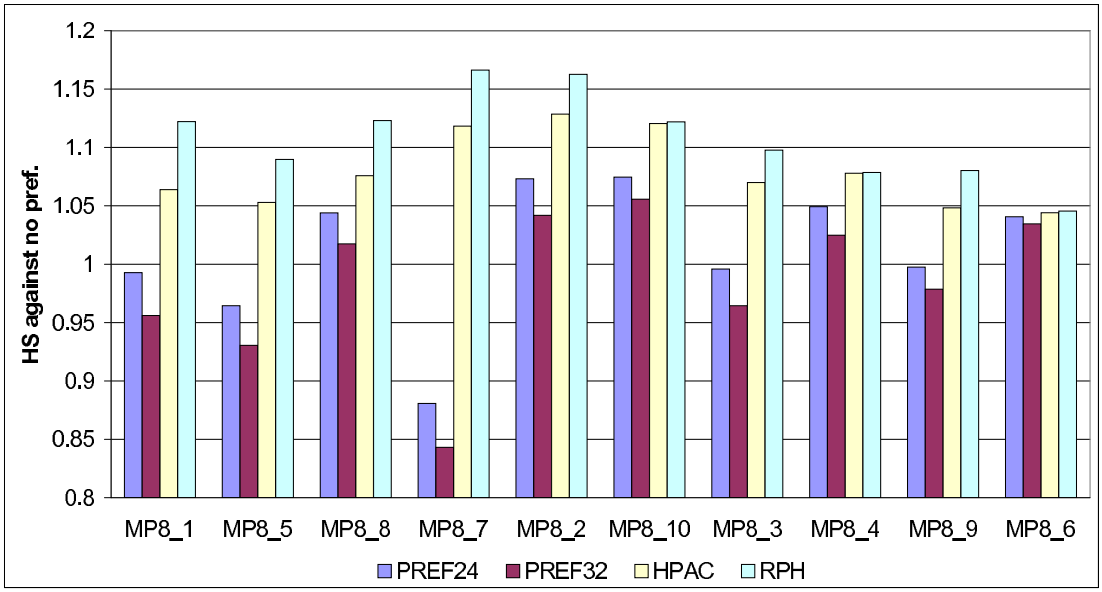


Figure 6.7: Prefetching performance of multi-programmed workloads.

From the multi-programmed workloads results we see that in general throttling gives the best performance results while the aggressive prefetching configuration *PREF32* is consistently the worst performer. In five cases (*MP8_1*, *MP8_3*, *MP8_5*, *MP8_7* and *MP8_9*) fixed degree prefetching actually decreases performance compared to a non-prefetching configuration. We evaluate the benefits of HPAC and RPH throttling using *PREF24* as our baseline. Conventional HPAC throttling increases on average the speedup of *PREF24* by 7.1%, while RPH increases the *PREF24* speedup by 10.9% on average, a 53.8% increase in speedup. In 7 out of the 10 benchmarks the RPH improves the performance increase of HPAC, sometimes by as much as 148% (*MP8_8*, 3.1% to 7.6%). In the remaining three benchmarks (*MP8_4*, *MP8_6* and *MP8_10*) both HPAC and RPH obtain similar speedups compared to *PREF24*.

The parallel benchmarks do not benefit as much from prefetch throttling. We attribute this to two main causes: a) the parallel benchmarks do not stress significantly the L2, due to increased data sharing, as explained in Section 6.5.2; and b) most parallel workloads are composed of similar threads that operate on different portions of data (i.e., threads are arranged basically in a SIMD mode of operation), which forces similar memory access patterns in all cores, leading to similar prefetch accuracy, interference and cache pollution metrics. This translates into a reduced exploration space where a prefetch throttling algorithm ends up effectively adjusting the aggressiveness of all prefetch engines at the same time. Even so, prefetch throttling has a significant per-

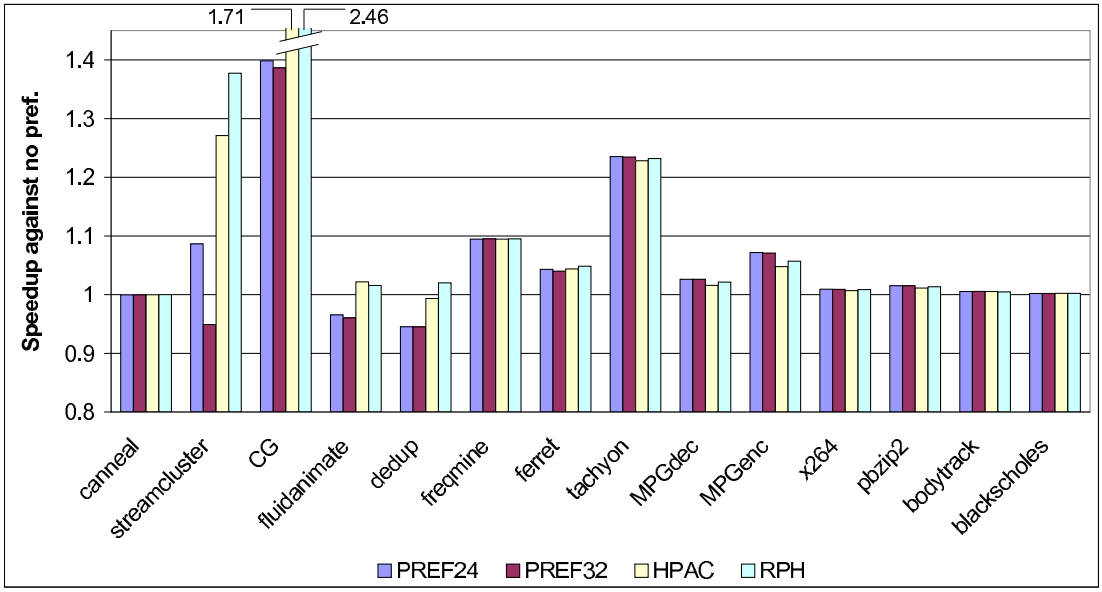


Figure 6.8: Prefetching performance of parallel benchmarks.

formance impact on three benchmarks: *dedup*, *streamcluster* and *CG*. In *dedup* HPAC increases the performance of *PREF24* by 5.1%, while RPH does so by 7.9%, a 55.2% increase in speedup. In *streamcluster* HPAC increases the performance of *PREF24* by 17% and RPH increases it by 26.8%, a speedup increase of 57.4%. Lastly, the biggest improvement of RPH over HPAC is in *CG*, where HPAC obtains an increase of 22.6% over *PREF24* while RPH increases the baseline performance by 76.1%, a 237% increase in speedup. On the other hand, RPH shows a slight decrease in performance in *fluidanimate*, where HPAC increases the speedup of *PREF24* by 5.9% and RPH does so by 5.2%. On average RPH increases the throttling benefits of HPAC by 22.6%

6.7 Bandwidth Usage

Figures 6.9 and 6.10 show the bus bandwidth increase due to the different prefetching strategies evaluated for multi-programmed and parallel workloads. For the multi-programmed workloads it can be seen that one of the main advantages of throttling is a considerable reduction in bandwidth usage. Both HPAC and RPH throttling achieve a similar reduction in bandwidth compared to fixed-degree prefetching configurations, with the only significant difference being in the parallel benchmark *dedup*, where HPAC shows a 48.3% increase in traffic (compared to a configuration with no prefetching) and RPH records a 19% increase.

In the parallel benchmarks the reduction in bus bandwidth is more moderate, with

the exception of *dedup*, *fluidanimate* and *streamcluster*. This is expected, since most parallel benchmarks evaluated are primarily CPU-bound.

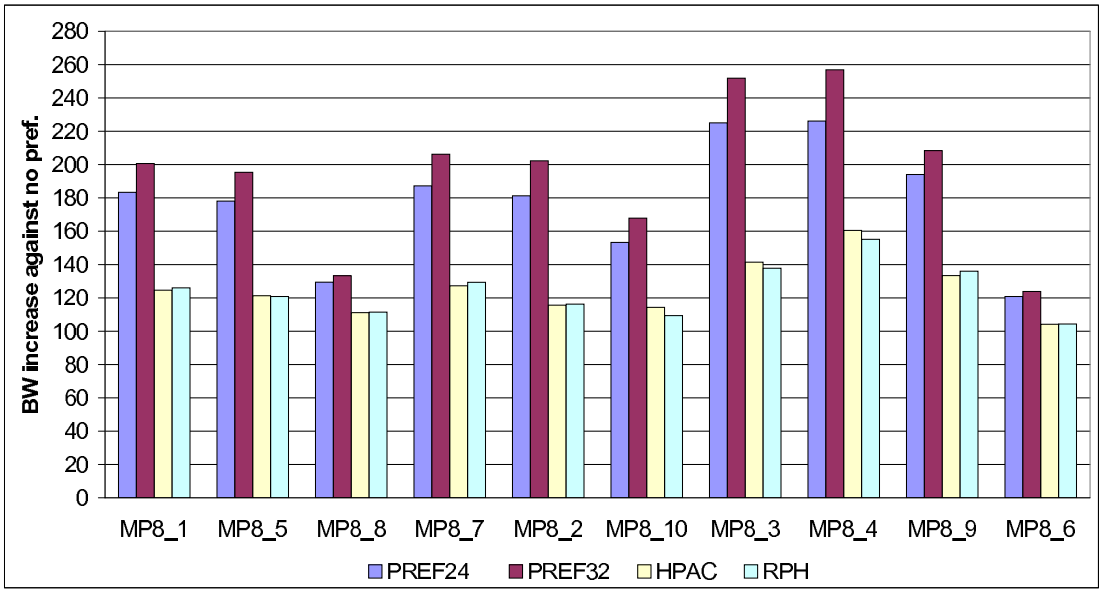


Figure 6.9: Prefetching bandwidth increase in multi-programmed workloads.

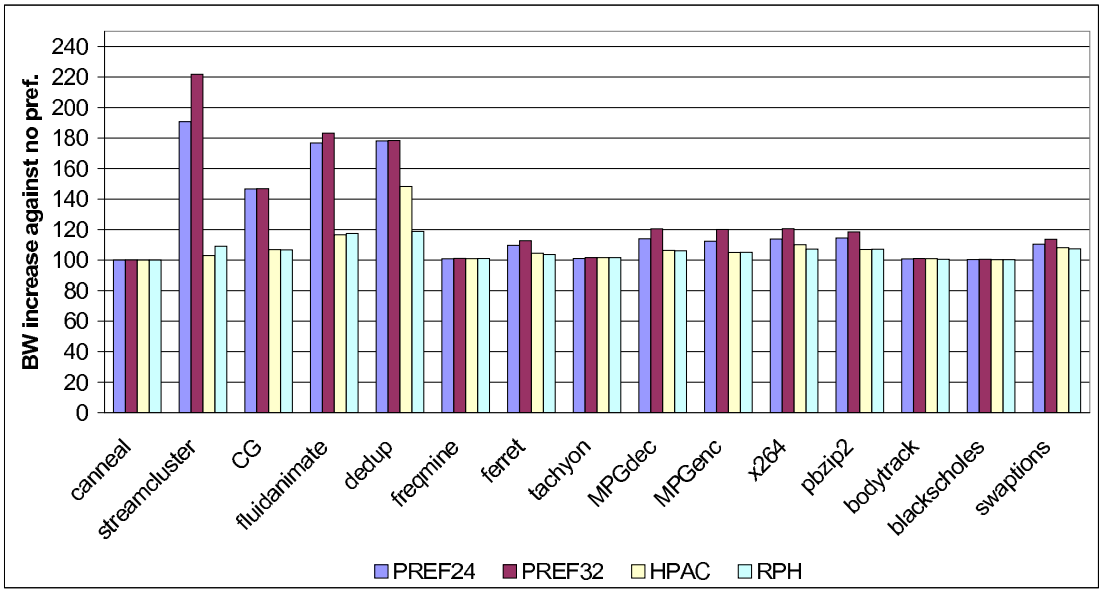


Figure 6.10: Prefetching bandwidth increase in parallel benchmarks.

6.8 Prefetch Fairness

In order to gain further insight into the advantages of prefetch throttling with RPHs over conventional HPAC we evaluate the prefetch fairness (Section 6.4) of both throt-

tling strategies. We evaluate the benchmarks used to produce the multi-programmed workloads, excluding *blackscholes*, *bodytrack* and *canneal*, whose non-prefetching performance is already under 1% of the performance obtained with a perfect L2 cache (CPU-bound benchmarks). Each benchmark is evaluated in 10 different multi-programmed configurations.

Figure 6.11 compares prefetch fairness of RPH against conventional HPAC throttling.

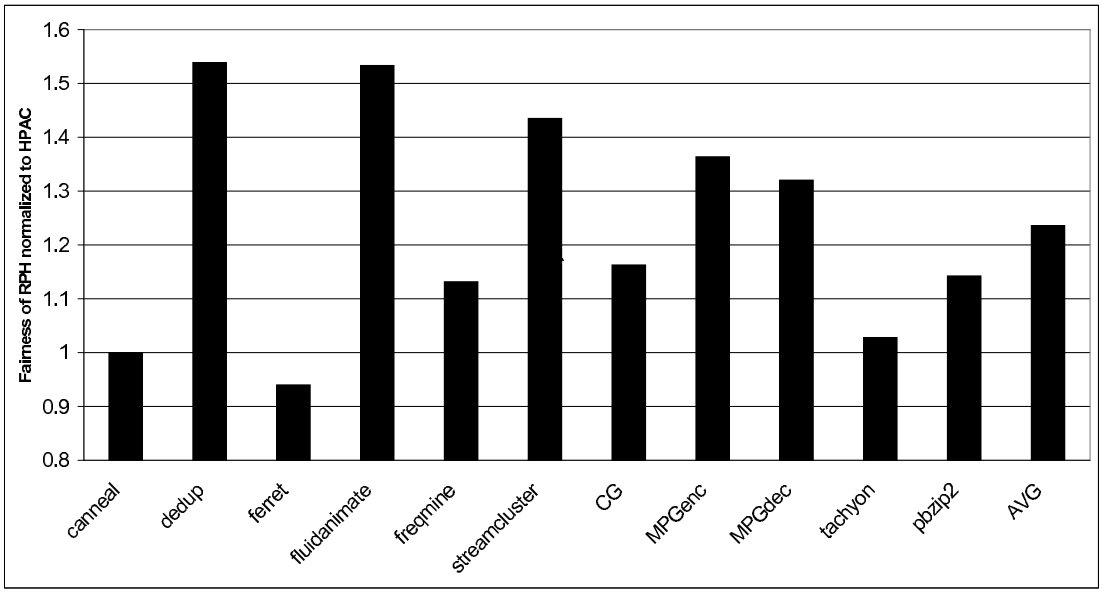


Figure 6.11: RPH throttling prefetch fairness normalized to HPAC prefetch fairness.

In table 6.3, we show an example fairness calculation for the benchmark *fluidanimate*. From those results, we see that fairness (the minus standard deviation of the performance variation in cycles) for this benchmark is -5597396 cycles when RPH is used, and -8523860 cycles when HPAC throttling is used. In Figure 6.11 we plot how much smaller the fairness metric is for HPAC compared to RPH; in the case of *fluidanimate* this number is 1.52, since $1.52 * (-5597396) = -8523860$.

From the results it can be seen that RPH throttling promotes a more fair dispatch of prefetches, with a 23% higher prefetch fairness on average than conventional HPAC throttling.

6.9 Influence of Adaptive Resizing

The two main techniques used in RPH throttling are prioritization of prefetch requests and adaptive resizing of the prefetch request queue. We quantify the effect of adaptive

resizing in the overall performance of the RPH. To do this we create a new prefetch throttling configuration called *PH* which essentially consists of an RPH without any adaptive resizing capabilities. Figures 6.12 and 6.13 show the performance of *PH* throttling compared to conventional HPAC throttling and RPH throttling.

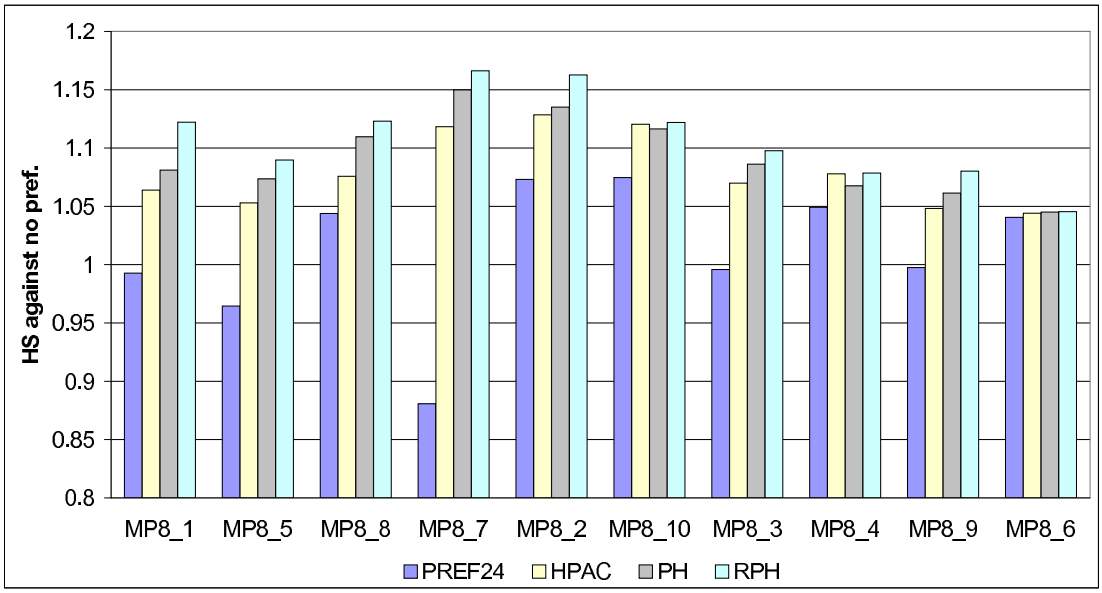


Figure 6.12: Results with and without adaptive resizing in multi-programmed workloads.

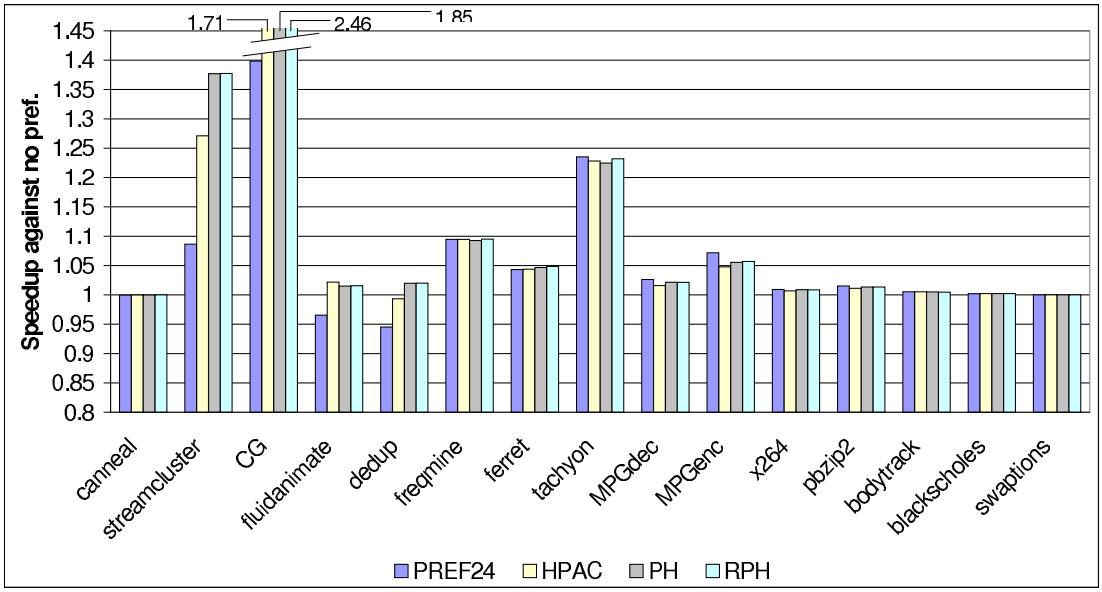


Figure 6.13: Results with and without adaptive resizing in parallel benchmarks.

The results show that the contribution of adaptive resizing to the overall RPH performance improvements varies by benchmark. In the multi-programmed scenario,

adaptive resizing contributes significantly in the overall performance of the RPH, and avoids performance degradation (compared to HPAC throttling) in two cases (*MP8_4* and *MP8_10*). In the parallel benchmarks the performance of *PH* is similar to the performance of *RPH*, with the exception of the *CG* benchmark. This is expected, since the parallel benchmarks exercise less memory traffic and therefore the need for global prefetch throttling is less pronounced.

6.10 Characterization RPH Queues

In this section we seek to gain further insight into the behavior of RPH PRQs by looking at how they behave

6.10.1 Average Number of Comparisons per Queue Operation

Figures 6.14 and 6.15 show the average number of comparisons needed to insert or extract an element from the RPH. As described in Section 5.4.1.1, to extract or insert an element from a binary heap, a number of comparison-swap steps over the heap array are performed sequentially. Furthermore, as explained in Section 5.4.1.2, optimized hardware implementations can perform one comparison-swap step per cycle. Therefore the number of comparisons needed to insert or extract an element into the RPH gives a good estimate of the overall run-time complexity of the queue operations for the set of benchmarks and workloads evaluated.

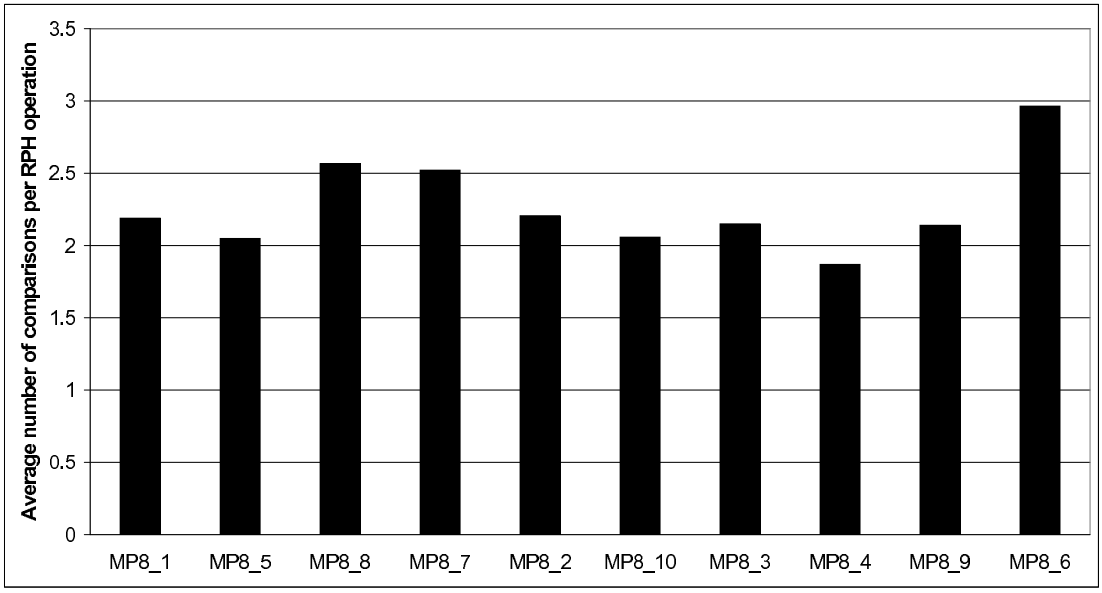


Figure 6.14: Average number of comparisons per RPH insertion/extraction operation in multi-programmed workloads.

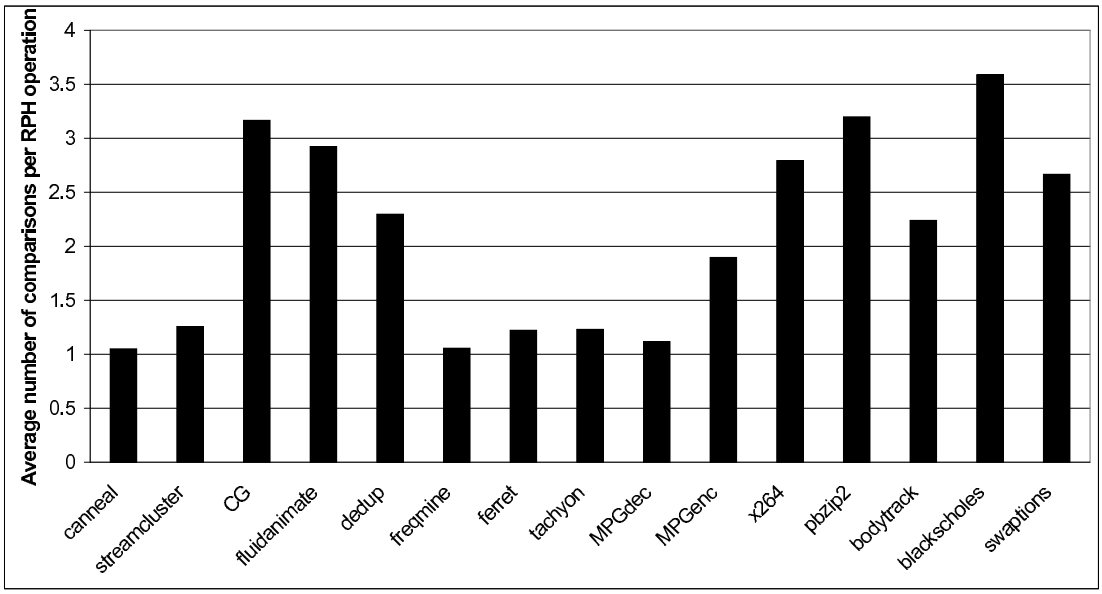


Figure 6.15: Average number of comparisons per RPH insertion/extraction operation in parallel benchmarks.

Recall that we use a 256-entry RPH queue for all the simulations (Table 6.1) and that both insertion and extraction from the RPH take a maximum of $log_2(256) = 8$ steps. However, for both multi-programmed and parallel benchmarks, the average number of comparisons needed per queue operation is much lower than the theoretical

maximum. This is expected. Firstly, the theoretical maximum number of comparison-swap steps only applies to the case of a full PRQ queue, situation that does not happen continuously. Secondly, the RPH resizes itself according to the utilization of the memory channel (Section 5.4.2.1), which involves a lower number of steps to insert and extract from the binary heap. Lastly, it can be shown that insertion operations in a binary heap can be considered to carry an average constant $O(1)$ time complexity, since most of the elements in the queue reside at the lower levels of the tree (Section 5.4.1.2).

6.10.2 Scaling States Histogram

Table 6.10.2 shows the histogram of time spent by the RPH in each of its four possible sizes state: full size queue, 50% size, 30% size and 1 element queue. The RPH scales to one of these states depending of the global usage of the memory channel (Chapter 5.4.2.1).

	RPH size			
Workload	**100%**	**50%**	**30%**	**1**
MP8_1	91.5	4.5	3.3	0.7
MP8_5	85.3	14.2	0.2	0.3
MP8_8	78.6	14.3	7.1	0
MP8_7	95.1	2.9	1.4	0.6
MP8_2	78.1	10.9	9.7	1.3
MP8_10	95.4	1.7	0.8	2.1
MP8_3	96.9	2.3	0.8	0
MP8_4	96.2	1.8	0.1	1.9
MP8_9	92.1	3	2.6	2.2
MP8_6	99.1	0.9	0	0

	RPH size			
Benchmark	**100%**	**50%**	**30%**	**1**
canneal	0	4.8	95.2	0
streamcluster	7.4	92.1	0	0.5
CG	80.3	12.7	4.1	2.9
fluidanimate	92.0	5.5	2.5	0
dedup	98.2	1.5	0.3	0
freqmine	97.4	0	2.6	0
ferret	62.5	29.2	0	8.3
tachyon	95.4	2.7	1.1	0.8
MPGdec	100	0	0	0
MPGenc	99.3	0.1	0.5	0.1
x264	89.1	5.1	2.1	3.7
pbzip2	86.7	6.7	6	0.6
bodytrack	100	0	0	0
blackscholes	100	0	0	0
swaptions	100	0	0	0

Table 6.4: Scaling states histogram for multi-programmed workloads (left) and parallel benchmarks (right)

On both multi-programmed workloads and parallel benchmarks, the majority of time is spent with the prefetching queue at its maximum size. However, most parallel benchmarks and all multi-programmed workloads resize the RPH a significant portion of the time. The exception to this behavior are the parallel benchmarks *MPGdec*,

bodytrack, *blackscholes* and *swaptions*, which are mostly CPU-bound (Section 6.5.1) and therefore do not saturate the memory channel.

In general, the more memory-bound a benchmark or a workload is, the more usage of the resizing capability of the RPH is observed. In the two most memory-bound parallel benchmarks (*canneal*, *streamcluster*), the majority of the time the queue stays in a resized state (30% for *canneal*, 50% for *streamcluster*). This indicates frequent periods of high memory activity. Multi-programmed workloads tend to have a more homogeneous behavior.

Comparing the data in table 6.10.2 with the performance obtained with and without adaptive resizing (Section 6.9), it is clear that although resizing is used frequently on both configurations, only the multi-programmed workloads benefit clearly from it on all situations. Multi-programmed workloads tend to have more continuous memory traffic than the parallel benchmarks. This is because data loading on parallel benchmarks tends to happen at the same time on all threads (due to the fact that they are usually composed of a collection of identically programmed threads), whereas on the multi-programmed workloads data loading and computation is usually interleaved across time, owing to its heterogeneous composition. Therefore, parallel benchmarks tend to do data loading in short, intense bursts, a situation not captured well by the resizing mechanism, as it bases its decisions on aggregating behavior in time windows. Additionally, with the exception of the top 3 benchmarks, parallel benchmarks are much less memory-bound than the multi-programmed workloads (Section 6.5.1).

Chapter 7

Summary of Contributions and Concluding Remarks

7.1 Summary of Contributions of this Thesis

In this dissertation we have explored several mechanisms to improve the efficiency of hardware data prefetchers. In the first part of this thesis we have studied in detail the behavior of state-of-the-art localizing prefetching algorithms. We have established that localizing prefetchers suffer from timeliness problems. This is because, in the process of localization, important chronological information that relates the order of activation of misses from different localized streams is lost. As a result, localizing prefetchers tend to issue prefetch requests too far in advance, resulting in decreased timeliness and increased cache pollution. To tackle this problem we propose the novel concept of Stream Chaining, a new level of correlation that allow prefetchers to reconstruct the chronological information lost during localization while at the same time filtering out spurious or infrequent misses. With Stream Chaining, the localized miss streams are linked in such a way that it is possible to reconstruct the *core flow* of misses in the application.

Based on the general concept of Stream Chaining, we propose a concrete implementation called Miss Graph prefetching. Miss Graph prefetching implements Stream Chaining using small weighted graphs that capture the core flow of miss stream activations in a manageable, low-complexity way. We implement Miss Graph prefetching in two popular localizing prefetchers: PC/DC and C/DC. We name the resulting new prefetchers PC/DC/MG and C/DC/MG, respectively. We provide implementation details for these new prefetchers. We use the Global History Buffer as the underly-

ing data structure to implement both chaining and non-chaining prefetchers. For the Stream Chaining prefetchers we show that the hardware modifications required are small and feasible to do with little added logic. We justify this by giving detailed storage and run-time complexity analysis. Finally we evaluate PC/DC/MG and C/DC/MG against their non-chaining counterparts. We show how these new prefetchers improve the timeliness, accuracy and coverage and therefore overall performance in most cases.

In the second part of this dissertation we have focused on prefetching in multi-core systems. We concentrate our research on the topic of prefetch throttling and arbitration, a relevant issue given the growing number of cores per chip. We have established that current prefetch throttling mechanisms either do not take advantage of the global metrics or, the few that do, do not leverage this advantage completely. We propose a new way of performing prefetch throttling at the PRQ (Prefetch Request Queue) level, with global knowledge of the metrics and state of each prefetching unit in the system. Our approach, which we call Resizable Prefetch Heaps (RPH), allows prioritization of prefetch requests based on their relative importance compared to other requests in the queue. To do so, we derive a formula that assigns a numeric priority to each prefetch request based on several local and global metrics. The RPH works as a priority queue, extracting at each time the request with the highest priority. Additionally, we allow the RPH to shrink and enlarge dynamically, depending on the utilization of the memory channel. In contrast with previous research on the topic, in our evaluation of RPH throttling we use both multi-programmed and parallel benchmarks. We evaluate the performance of fixed degree aggressive prefetching, a state-of-the-art conventional throttling algorithm (HPAC) and RPH throttling. We introduce a metric for prefetch fairness, which relates the variance in performance of a prefetcher in the presence of other competing prefetchers. We show how RPH throttling improves the performance of HPAC in both multi-programmed workloads and parallel benchmarks.

7.1.1 Future Work

Since both main contributions presented in this thesis are based in new heuristics, there is ample space for further research into them. Regarding Stream Chaining, the two main areas that are more likely to produce interesting research results are new heuristics for chaining streams and new prefetch issuing policies (i.e., how to navigate the graph of linked streams and how many prefetches to issue per stream). For the former, further research into modelling the core flow of misses in an application is needed, as

the key for successful Stream Chaining lies into correctly identifying the core repetitions in miss stream activations. For the latter, research into more adaptive prefetch issuing techniques is likely to improve even more the timeliness of Stream Chaining prefetchers. Although more complex heuristics are promising, one of the main challenges future research will likely face is keeping the run time of such algorithms within the timing constraints of the cache memory.

Regarding Resizable Prefetch Heaps, considerable further research can be aimed towards priority assignment formulas. Resizable Prefetch Heaps are versatile and can simulate a wide range of queuing behaviors with just modifying the priority assignment formula. This opens the door to several interesting research scenarios, such as Quality of Service support, reconfigurable prefetch assignment formulas and integration into the operating system process prioritization scheme. Furthermore, the resizing algorithm itself is subject to improvements and more research. Resizing the prefetch request queue acts as a gradual global throttle to prefetching in the whole system. Further research could tackle the problem of understanding when is it more beneficial to adjust this global throttle instead of limiting each prefetcher individually. As with Stream Chaining, new research should also take into account the hardware timing constraints. In this area, future research could focus on new priority queue implementations specially designed for prefetch request queues.

7.2 Concluding Remarks

The memory wall is a well known performance limitation that affects past and current hardware architectures. In the past, the memory wall was specially obvious due to the great disparity in growth between processor and main memory operating frequencies. Nowadays, this trend for ever-increasing processor speeds has been exchanged for an increasing number of cores per processor. This continues to exacerbate the memory wall problem, since now several cores have to compete for memory access. Therefore, although the memory wall is now an old problem, it is becoming clear that there will be no immediate absolute remedy for it in the short-medium term.

Hardware prefetching is a proven technique to alleviate the effects of the memory wall. On one hand, hardware prefetching, as opposed to software prefetching, has the advantage of being able to access run-time information of the program, as well as being universally applicable without the need of recompilation. On the other hand, it is limited by the scope and complexity of the prediction heuristics that can be feasibly

implemented in hardware. Hardware prefetching at the last level cache (i.e., the last on-chip cache) is specially effective, since this is where the latency disparity between a miss and a hit is the greatest.

In the design of hardware prefetching algorithms there is an inherent performance/-cost trade-off. Trivial or very simple prefetching algorithms can be implemented with minimal hardware modifications and cost. However such prefetchers will not obtain significant performance benefits on any but the simplest memory access patterns. By contrast, several very sophisticated algorithms have been proposed previously in the literature. These algorithms, capable of tracking complicated data structure access patterns such as the transversal of linked lists, come at the cost of complicated hardware implementations and huge storage overheads. Traditionally, the industry has favored prefetching algorithms closer to the simple but easy to implement end of the spectrum, opting to use the increasing number of transistors yielded by improved manufacturing technology to build bigger cache memories. Academia, on the other hand, has typically concentrated on the more sophisticated, but not necessarily practical, type of prefetching algorithms. A medium ground must be found that favors the development of more efficient and complex prefetching methods, while at the same time making sense from the return-on-investment perspective. Throughout this dissertation I have focused on technologies and algorithms that I believe lay in this middle ground.

Another pressing issue in the research of hardware prefetching algorithms has been the move to multi-core architectures. The problem of prefetcher coordination in such environments has become, in my opinion, as important as the development of new prefetching heuristics. The effects of a badly behaving prefetching engine on just a single core can degrade the performance of the whole system. This is even more relevant as we are - slowly but surely - migrating from a traditionally sequential programming perspective to a multi-threaded/multi-programmed throughput-computing paradigm. In this scenario, obtaining modest but consistent performance improvements over the whole system will become clearly much more valuable than achieving irregular high sequential speedups.

Appendix A

Benchmark Descriptions

A.1 SPEC CPU2006 benchmarks

SPEC CPU is a collection of CPU and memory intensive benchmarks from the Standard Performance Evaluation Corporation (SPEC), a non-profit organization formed by hardware and software vendors, universities, research institutions and other representatives of industry and academia. SPEC periodically releases updated versions of its benchmark suite, named after the year the release was made. In this dissertation we evaluate a selection of benchmarks from the latest released version: SPEC CPU2006.

All SPEC CPU benchmarks are written in C, C++ or Fortran. Benchmarks are typically broadly classified according to the functional unit they stress the most (integer or floating point). SPEC CPU does not include synthetic benchmarks, and therefore all its benchmarks come from applications and workloads found in the real world. Each benchmark consists of a customized (and usually stripped-down) version of a program representative of workloads found in scientific or commercial computing.

Below is a list of all the SPEC benchmarks used for the evaluation, along with a brief description of their purpose and the data used in the reference input. We list for each benchmark its SPEC code as well as its mnemonic.

A.1.1 Integer Benchmarks.

400.perlbench: Workload consisting of a cut-down version of the popular Perl interpreter (v. 5.8.7). This benchmark also includes several third party modules. The main part of the workload performs text processing and parsing (spam filtering, HTML parsing, message-digest calculation, string manipulation, etc.). Source code written in

C.

401.bzip2: Benchmark for performing lossless data compression. This benchmark consists of a modified version of Julian Seward's *bzip2* program. All processing is done in main memory and no file I/O is done other than reading the file at the beginning of the execution. The reference input set contains several kinds of data files: JPEG images, a program binary, program source code and HTML text. Source code written in C.

429.mcf: Combinatorial optimization program. This workload is a streamlined version of the MCF program by Andreas Löbel. MCF optimizes single depot vehicle scheduling in public mass transport systems. It implements the well-known simplex algorithm. The reference input set contains a large vehicle scheduling problem. Source code written in C.

445.gobmk: Artificial intelligence / Game theory simulation. This benchmark evaluates and plays several games of Go. Based on the popular GNU Go engine. The reference input file contains a series of Go game descriptions and several commands to evaluate and play the next move. Source code written in C.

458.sjeng: Artificial intelligence / Game theory simulation. This workload is based on the Sjeng chess engine v. 11.2. It uses a combination of game tree search (alpha-beta search with several pruning and priority heuristics) and pattern recognition techniques to evaluate several chess moves. The code has been modified for SPEC to better reflect the workloads found nowadays in game theory applications. Source code written in C.

462.libquantum: Quantum computing simulation benchmark. This workload is implemented using the libquantum library by Björn Butscher and Hendrik Weimer. libquantum is a library to simulate quantum computers. The reference input set simulates the Shor's factoring algorithm for quantum machines. Source code written in C.

464.h264ref: Video compression benchmark based on the reference H.264 video codec implementation by Karsten Sühring *et al*. The benchmark compresses video to the H.264 format. The reference input set contains two uncompressed sequences, one from real life video at 176x144 resolution and another from a video game at 520x320 resolution. Source code written in C.

471.omnetpp: Benchmark that performs network simulations. This workload is based on the OMNeT++ discrete event simulation system by András Varga and Omnest Global Inc. The reference workload simulates a large Ethernet backbone with several

ancillary LANs connected to it. The model simulated contains about 8000 computers, 900 switches and hubs and several Ethernet technologies (10baseT, 100MB half/full duplex, Gigabit, etc.). Source code written in C++.

A.1.2 Floating Point Benchmarks

433.milc: Physics simulation program, for use in quantum chromodynamics. This workload is a serial version of the su3imp program by Steven Gottlieb from Indiana University. This code is in use extensively (millions of node-hours) at the United States Department of Energy and National Science Foundation supercomputers. All the input sets refer to the same problem, with different grid sizes. Source code written in C.

434.zeusmp: Computational fluid dynamics code based on ZEUS-MP, developed by the Laboratory for Computational Astrophysics (part of NCSA) at the University of Illinois - Urbana Champaign. The reference input set solves a 3D blast-wave simulated along the presence of a magnetic field. Source code Written in Fortran 77.

435.gromacs: Molecular dynamics chemistry simulation workload. Derived from the popular molecular dynamics package GROMACS. All SPEC input sets simulate the same scenario: the protein Lysozyme in a solution of water and ions, with the only difference being the number of simulation steps performed (6000 for the reference input set). Source code written mostly in C, with the inner loop computation written in Fortran 77.

444.namd: This workload isolates the serial inner loop of the NAMD parallel program, used for the simulation of large biomolecular systems. All the input data sets use the same simulation scenario, with the only difference being the number of simulation iterations (38 for the reference input set). Source code written in C++.

447.dealII: Partial differential equation solver using the Adaptive Finite Element Method. This benchmark uses the deal.II library of equation solvers, which in turn uses state of the art C++ programming paradigms and techniques, including the popular Boost library of data structures and algorithms. The reference input data set is generated on the fly by the program, and solves a Helmholtz-type equation, which often arises in the study of physics problems that involve partial differential equations. Source code written in C++.

450.soplex: Linear programming workload, based on the SoPlex program version 2.1 by Roland Wunderling, Thorsten Koch and Tobias Achterberg. It uses the Simplex Method to solve a linear programming problem. Due to the nature of the problem, sev-

eral computational algebra algorithms for sparse matrices are used. In particular sparse LU factorization and algorithms for triangular equation systems are used extensively. The reference input set uses the test case “rail2586” from the netlib package. Source code written in C++

470.lbm: Computational fluid dynamics benchmark based on the code written by Thomas Pohl. This program implements the “Lattice Boltzmann Method” (LBM) for simulation of incompressible fluids. The reference input data set simulates the shear flow driven by a “sliding wall” boundary condition for 3000 timesteps. Source code written in C.

482.sphinx3: Speech recognition benchmark based on the popular Sphinx-3 speech recognition package from Carnegie Mellon University. The reference input set contains several audio files in raw format to be processed by the speech recognition engine. Source code written in C.

A.2 BioBench Benchmarks

BioBench [32] is a benchmark suite for Bioinformatics applications. Bioinformatics is a composite research field that encompasses Informatics, Biology and Medicine. It uses computationally intensive techniques to gain better understanding of biological and biochemical processes. Data-mining, pattern recognition and machine learning techniques are commonly used in Bioinformatics, with the difference that the backing database from which they operate models some kind of biological process or structure.

Biobench was created as the result of a collaboration of the University of Maryland with Intel Corporation in 2006. It consists of a representative set of data-mining algorithms and applications currently relevant in the field of Bioinformatics.

Due to limitations in our simulation environment, it was not possible to compile the *BLAST* and *mummer* benchmarks. The rest of the BioBench workloads are described below.

clustalw: Multiple sequence alignment benchmark based on the CLUSTAL package. Multiple sequence alignment is the process of aligning more than two nucleotide sequences to find regions of similarity. We use the input data set from the benchmark *hmmer*, described below. Source code written in C.

tiger: This workload is the TIGR assembler suite v.2 from the Institute for Genomic Research, Rockville. Sequence assembly is a technique used to generate full sequence data from small overlapping partial sequences produced by DNA sequencing

hardware. The reference input file contains 24190 partial RNA sequences from *Picea sitchensis*. Source code written in C.

hmmer: Sequence profile search benchmark based on the HMMER software from the Washington University in St. Louis. This workload uses Hidden Markov Models (HMM) to search for similarities in a DNA database. The reference input data searches a collection of small protein sequences against the SwissPROT protein database. Source code written in C.

phylip: Phylogenetic analysis benchmark based on the PROTPARS tool from the PHYLIP software package (University of Washington). Phylogenetic analysis tries to find out how a group of related protein sequences were derived from a common ancestor. In order to do this, the program uses a hierarchical data structure called the Phylogenetic tree. Source code written in C.

fasta: Sequence similarity search based on University of Virginia's FASTA suite v.3.4t21. Similarity search looks for similarities between DNA or protein sequences, or search for certain subsequences in large sequence databases. The reference input data set consists of two databases: a 170MB DNA database from NCBI GeneBank and the entire SwissPROT protein database (70MB), along with their corresponding search sequences. Source code written in C.

A.3 Parallel Benchmarks

For the evaluation of Resizable Prefetch Heaps (Section 6) we use benchmarks from the PARSEC [33] and ALPBench [34] benchmark suites, as well as two stand-alone parallel programs: *pbzip2*, a parallel compression program that processes files using the popular *bzip2* compression algorithm and *CG*, a synthetic scientific workload.

For each benchmark we give a short description about its purpose, a description of the input data set used, how the program was parallelized and what language is the source code written in. Most benchmarks were parallelized *explicitly*, that is, with the explicit creation of threads and use of (POSIX) synchronization primitives. We include, however, three benchmarks which were parallelized *implicitly* with the use of OpenMP directives (*bodytrack*, *freqmine* and *CG*), and one benchmark that, while using standard thread creation primitives, it uses atomic operations and therefore has no synchronization (*canneal*).

A.3.1 PARSEC benchmarks

Due to limitations in our simulation environment, the PARSEC benchmarks *facesim*, *raytrace* and *vips* could not be cross-compiled. This is because these benchmarks have strong dependencies to external libraries that were impossible to cross-compile or mock, such as the X-Windows library.

We use the *simlarge* input data sets, the largest ones that can be feasibly be used for architectural simulation.

blackscholes: Financial benchmark from the Intel Financial Services Application Benchmarks. Computes the prices of a portfolio of stock options using the Black-Scholes partial differential equation. The reference input data consists of 65,536 options, which are loaded into memory before any computation starts. Explicit parallelization with standard synchronization primitives. Source code written in C.

bodytrack: Computer vision benchmark from the Intel RMS (Recognition, Mining and Synthesis) program [36]. Tracks the pose of a tracker-less human body in 3D, using an annealed particle filter to detect edges and the body silhouette. The input data set consists of 4 frames from 4 cameras, 4,000 particles and 5 annealing layers. Parallelized with OpenMP directives. Source code written in C++.

canneal: Chip routing benchmark. Developed by Princetown University, it uses cache-aware simulated annealing to optimize the routing of a chip design. The algorithm employed performs random swaps between chip elements and evaluates the resulting routing. The input data sets optimizes routing in a 400,000 netlist, performing 15,000 swaps per temperature step and starting with a temperature of 2,000°. This benchmark uses fine grained parallelism, performing element swaps atomically and in a lock-free manner. Source code written in C++.

dedup: Deduplication and compression benchmark, based on a kernel developed by Princetown University. Deduplication is a method used in backup and large storage systems where multiple copies of data are replaced by references to an unique copy. The reference input data set consist of an archive of 184MB, which contains diverse types of files. Parallelized explicitly with standard synchronization primitives Source code written in C.

ferret: Benchmark for content-based similarity search in large, feature-rich multimedia databases. Based on the *Ferret* toolkit, developed by Princetown University. The reference input data set consists of a database of 34,973 images, on which 256 queries to find the top 10 most similar images are done. Explicitly parallelized with

standard synchronization primitives. Source code written in C.

fluidanimate: Fluid dynamics simulation benchmark part of Intel RMS suite. The input data set consists of 300,000 particles, which are simulated for 5 frames. Parallelized explicitly with standard synchronization primitives. Source code written in C++.

freqmine: Data mining benchmark originally developed by Concordia University. This benchmarks implements *Frequent Itemset Mining*, which is the basis of *Association Rule Mining*, a common data mining problem which aims to learn relations between variables in large databases. Association Rule Mining is used in diverse fields such as bioinformatics, financial data mining or log analysis. The input data set consists of an anonymized webserver logfile from a Hungarian news portal, containing 990,000 click streams. Parallelized with OpenMP directives. Written in C++.

streamcluster: Computing kernel developed by Princetown University to solve the *online clustering* problem: for a stream of input points, find a pre-established number of median points in such a way that every point of the stream ends up associated to its nearest median point. The input data set consists of 16,384 128-dimensional points, for which 10 to 20 median points are sought. Parallelized explicitly with standard synchronization primitives. Source code written in C++.

swaptions: Financial analysis benchmark part of the Intel RMS workloads . It applies the Heath-Jarrow-Morton framework to set the price to a portfolio of swaptions. A swaption is a type of financial option which grants its owner the right to perform a financial swap operation. The input data set consists of 64 swaptions, on which 20,000 simulations are performed. Parallelized explicitly with standard synchronization primitives. Source code written in C++.

x264: Parallel H.264 video encoder. The input data set is a 640×360 pixels movie with 128 frames. Parallelized explicitly with standard synchronization primitives. Source code written in C++.

A.3.2 ALP Benchmarks

The ALPBench[34] benchmark suite is a collection of multimedia-oriented parallel benchmarks developed by the University of Illinois at Urbana-Champaign with support from Intel, AMD and the National Science Foundation. All benchmarks are explicitly parallelized using standard POSIX synchronization primitives. Due to limitations in our simulation environment, the *SpeechRec* benchmark could not be compiled.

MPGdec: MPEG-2 decoding benchmark. This is a parallel version of the reference implementation provided by the MPEG Software Simulation Group (MSSG). The input data set consists of a HDTV 1440 $\times$ 1080 pixels public domain video stream (`http://www.archive.org/details/ligouHDR-HC1_japan`). Parallelized explicitly with standard synchronization primitives. Source code written in C.

MPGenc: MPEG-2 encoding benchmark. Like *MPGdec*, this is a parallelized version of the original reference implementation provided by the MSSG. The input data set is the decoded stream used in the *MPGdec* benchmark. Source code written in C.

Raytrace/tachyon: Ray-tracing benchmark. This benchmark is the Tachyon parallel raytracer (`http://jedi.ks.uiuc.edu/~johns/raytracer/`) unmodified. The input data set is the sample input file (bundled with the source code) `820spheres.dat`, concatenated 35 times for a total of 28,700 objects to render. Parallelized explicitly with standard synchronization primitives. Source code written in C.

A.3.3 Standalone programs

CG: Synthetic scientific benchmark part of NASA's NAS parallel benchmarks suite [35]. Computes the conjugate gradient of a given matrix. The input used is the "C" synthetic data set. Parallelized implicitly with OpenMP directives. Source code written in C.

pbzip2: Parallel compression program that uses the popular *bzip2* algorithm. To parallelize the compression, the input data is divided across the threads. Synchronization in the master thread enforces that the compressed data is written back in the correct order. The input data to compress is the same used by the *401.bzip2* SPEC2006 benchmark (Section A.1.1). Parallelized explicitly with standard synchronization primitives. Source code written in C.

Bibliography

[1] J. W. J. Williams. “Algorithm 232 Heapsort”. *Communications of the ACM*, Vol 7(6), pages 378-348, June 1964.

[2] N. Jouppi. “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.” *Intl. Symp. on Computer Architecture*, pages 364-373, May 1990.

[3] S. Palacharla and R. E. Kessler. “Evaluating Stream Buffers as a Secondary Cache Replacement.” *Intl. Symp. on Computer Architecture*, pages 24-33, May 1994.

[4] W. Anacker and C. PẆang. “Performance Evaluation of Computing Systems with Memory Hierarchies.” *IEEE Transactions on Computers*, Vol EC-16, No 6, pp764-774, Dec 1967.

[5] A.J. Smith. “Sequential Program Prefetching in Memory Hierarchies”. *IEEE Transactions on Computers*, Vol 11, No 12, pp7-21, Dec 1978.

[6] J. W. C. Fu, J. H. Patel, and B. L. Janssens. “Stride Directed Prefetching in Scalar Processors.” *Intl. Symp. on Microarchitecture*, pages 102-110, December 1992.

[7] W.Y. Chen, S.A. Mahlke, P.P. Chang and W-M. W. Hwu. “Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching.” *Intl. Symp. on Microarchitecture*, 1991.

[8] W-F. Lin, S. K. Reinhardt and D. Burger. “Reducing DRAM Latencies with an Integrated Memory Hierarchy”. *Intl. Symp. On High Performance Computer Architecture*, Jan 2001.

[9] G. B. Kandiraju and A. Sivasubramaniam. “Going the Distance for TLB Prefetching: An Application-Driven Study.” *Intl. Symp. on Computer Architecture*, pages 195-206, May 2002.

[10] K. J. Nesbit and J. E. Smith. “Data Cache Prefetching Using a Global History Buffer.” *Intl. Symp. on High Performance Computer Architecture*, pages 96-106, February 2004.

[11] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. “AC/DC: An Adaptive Data Cache Prefetcher.” *Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 135-145, September 2004.

[12] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. “Temporal Streaming of Shared Memory.” *Intl. Symp. on Computer Architecture,* pages 222-233, June 2005.

[13] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. “Spatial Memory Streaming.” *Intl. Symp. on Computer Architecture,* pages 252-263, June 2006.

[14] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi. “Spatio-Temporal Memory Streaming.” *Intl. Symp. on Computer Architecture,* June 2009.

[15] W. A. Wulf, S. A. McKee. “Hitting the Memory Wall: Implications of the Obvious.” Computer Architecture News, 1995.

[16] S. A. McKee. “Reflections on the memory wall.” *Conf. on Computing Frontiers,* 2004.

[17] R. Cooksey, D. Colarelli and D. Grunwald. “Content-Based Prefetching: Initial Results.” *Lecture Notes in Computer Science,* pages 33-55, 2001.

[18] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y. Lee, D. Lavery and J. P. Shen. “Speculative precomputation: long-range prefetching of delinquent loads”. *Intl. Symp. on Computer Architecture,* pages 14-25, 2001.

[19] H. C. Young and E. J. Shekita. “An intelligent I-cache prefetch mechanism”. *IEEE International Conference on Computer Design,* pages 44-49, 1993.

[20] , R. HṖatterson and G. A. Gibson. “Exposing I/O concurrency with informed prefetching”. *Third International Conference on Parallel and Distributed Information Systems,* pages 7-16, September 1994.

[21] F. Dahlgren, M. Dubois and P. Stenström. “Sequential hardware prefetching in shared-memory multiprocessors” . *IEEE Transactions on Parallel and Distributed Systems,* Vol 6, No. 7, pages 733-746, 1995.

[22] I. Hur and C. Lin. “Memory Prefetching Using Adaptive Stream Detection”. *Intl. Symp. on Microarchitecture,* 2006.

[23] S. Srinath, O. Mutlu, H. Kim and Y. N. Patt. “Feedback Directed Prefetching: Improving the Performance and Bandiwdth-Efficiency of Hardware Prefetchers”. *Intl. Symp. on High Performance Computer Architecture,* 2007.

[24] E. Ebrahimi, O. Mutlu, C. J. Lee and Y. N. Patt. “Coordinated Control of Multiple Prefetchers in Multi-Core Systems.” *Intl. Symp. on Microarchitecture,* pages 12-16, December 2009.

[25] V. Srinivasan, G.S. Tyson and E.S. Davidson. “A Static Filter for Reducing Prefetch Traffic.” *Technical Report CSE-TR-400-99, Univ. of Michigan.* 1999.

[26] W-F. Lin, S. K. Reinhardt, D. Burger and T. R. Puzak. “Filtering Superfluous Prefetches using Density Vectors.” *Intl. Conf. on Computer Design,* pages 124-132, 2001.

[27] X. Zhuang and H-H. S. Lee. “Reducing Cache Pollution via Dynamic Data Prefetch Filtering.” *IEEE Transactions on Computers,* vol. 56, no. 1, January 2007.

[28] S. Kim, D. Chandra and Y. Solihin. “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture.” *International Conference on Parallel Architectures and Compilation Techniques,* September 2004.

[29] P. Díaz and M. Cintra. “Stream Chaining: Exploiting Multiple Levels of Correlation in Data Prefetching.” *Intl. Symp. on Computer Architecture,* June 2009.

[30] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator. http://sesc.sf.net

[31] Various Authors. “SPEC CPU2006 Benchmark Descriptions”. *ACM SIGARCH Computer Architecture News,* Vol 35, No. 1, March 2007.

[32] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. “BioBench: A benchmark suite of bioinformatics applications.” *Intl. Symp. on Performance Analysis of Systems and Software,* pages 2-9, March 2005.

[33] C. Bienia, S. Kumar, J. P. Singh and K. Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications.” *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques,* October 2008.

[34] M. L. Li, R. Sasanka, S. V. Adve, Y. K. Chen, E. Debes. “The ALPBench Benchmark Suite for Complex Multimedia Applications.” *Proceedings of the IEEE International Symposium on Workload Characterization,* October 2005.

[35] D. Bailey *et al.* “The NAS parallel benchmarks.” NASA Technical Report RNR-94-007. 1994.

[36] P. Dubey. “A Platform 2015 Workload Model. Recognition, Mining and Synthesis Moves Computers to the Era of Tera”. Intel Corporation White Paper. 2005.

[37] A. Ioannou, M. Katevenis. “Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High Speed Networks”. IEEE/ACM Transactions on Networking (TON), Volume 15, issue 2. April 2007.

[38] K. E. Batcher. “Sorting networks and their applications”. Proceedings of the AFIPS Spring Joint Computer Conference 32, pp. 307-314, 1968.

[39] Oberlin, S., R. Kessler, S. Scott and G. Thorson. “The Cray T3E Architecture Overview.” Cray Research Inc., Eagan, MN, 1996.

[40] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. “POWER4 system microarchitecture”. IBM Technical White Paper, Oct. 2001.

[41] J. Clabes *et al.* “Design and implementation of the Power5 microprocessor”. Proceedings of the 41st Annual Conference in Design Automation. pp. 670-672. 2004.

[42] J. Doweck. “Inside Intel Core Microarchitecture and Smart Memory Access.” White paper, Intel Corporation, 2006. http://download.intel.com/technology/architecture/sma.pdf.

[43] R. Kalla, B. Sinharoy, and J. M. Tendler. “IBM POWER5 Chip: A Dual-Core Multithreaded Processor.” *IEEE Micro,* vol. 24, no. 2, pages 40-47, March-April 2004.

[44] J.-L. Baer and T. F. Chen. “An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty.” *Supercomputing Conf.,* pages 176-186, November 1991.

[45] Z. Hu, S. Kaxiras, and M. Martonosi. “Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior.” *Intl. Symp. on Computer Architecture,* pages 209-220, May 2002.

[46] Z. Hu, M. Martonosi, and S. Kaxiras. “TCP: Tag Correlating Prefetchers.” *Intl. Symp. on High Performance Computer Architecture,* pages 317-326, February 2003.

[47] D. Joseph and D. Grunwald. “Prefetching Using Markov Predictors.” *Intl. Symp. on Computer Architecture,* pages 252-263, June 1997.

[48] A.-C. Lai, C. Fide, and B. Falsafi. “Dead-Block Prediction & Dead-Block Correlating Prefetchers.” *Intl. Symp. on Computer Architecture,* pages 144-154, June 2001.

[49] H. Q. Le, W. J. Starke, J. Stephen Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. “IBM POWER6 Microarchitecture.” *IBM Journal of Research and Development,* vol. 51, no. 6, pages 639-662, November 2007.

[50] D. G. Pérez, G. Mouchard, and O. Temam. “MicroLib: A Case for the Quantitative Comparison of Micro-Architecture Mechanisms.” *Intl. Symp. on Microarchitecture,* pages 43-54, December 2004.

[51] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. WRL Research Report, 2001/2.

[52] Y. Solihin, J. Lee, and J. Torrellas. “Using a User-Level Memory Thread for Correlation Prefetching.” *Intl. Symp. on Computer Architecture,* pages 171-182, May 2002.

[53] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. “Branch History Guided Instruction Prefetching.” *Intl. Symp. on High Performance Computer Architecture,* pages 291-300, January 2001.

[54] W. Zhang, B. Calder, and D. M. Tullsen. “A Self-Repairing Prefetcher in an Event-Driven Optimization Framework.” *Intl. Symp. on Code Generation and Optimization,* pages 50-64, March 2006.

[55] S. P. Vanderwiel and D. J. Lilja. “Data Prefetch Mechanisms.” *ACM Computing Surveys,* vol. 32, no. 2, June 2000.

[56] H. Zhu, Y. Chen, X. Sun. “Timing local streams: improving timeliness in data prefetching.” *Intl. Conference on Supercomputing,* 2010.