REFERENCE SPECULATION-DRIVEN MEMORY MANAGEMENT

A Dissertation

by

JINCHUN KIM

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Paul V. Gratz
Committee Members,	Daniel A. Jiménez
	A. L. Narasimha Reddy
	Jean-Francois Chamberland
Head of Department,	Miroslav M. Begovic

May 2017

Major Subject: Computer Engineering

ABSTRACT

The "Memory Wall", the vast gulf between processor execution speed and memory latency, has led to the development of large and deep cache hierarchies over the last twenty years. Although processor frequency is no-longer on the exponential growth curve, the drive towards ever greater main memory capacity and limited off-chip bandwidth have kept this gap from closing significantly. In addition, future memory technologies such as Non-Volatile Memory (NVM) devices do not help to decrease the latency of the first reference to a particular memory address. To reduce the increasing off-chip memory access latency, this dissertation presents three intelligent speculation mechanisms that can predict and manage future memory usage.

First, we propose a novel hardware data prefetcher called Signature Path Prefetcher (SPP), which offers effective solutions for major challenges in prefetcher design. SPP uses a compressed history-based scheme that accurately predicts a series of long complex address patterns. For example, to address a series of long complex memory references, SPP uses a compressed history signature that is able to learn and prefetch complex data access patterns. Moreover, unlike other history-based algorithms, which miss out on many prefetching opportunities when address patterns make a transition between physical pages, SPP tracks the stream of data accesses across physical page boundaries and continues prefetching as soon as they move to new pages. Finally, SPP uses the confidence it has in its predictions to adaptively throttle itself on a per-prefetch stream basis. In our analysis, we find that SPP outperforms the state-of-the-art hardware data prefetchers by 6.4% with higher prefetching accuracy and lower off-chip bandwidth usage.

Second, we develop a holistic on-chip cache management system that tightly integrates data prefetching and cache replacement algorithms into one unified solution. Also,

we eliminate the use of Program Counter (PC) in the cache replacement module by using a simple dead block prediction with global hysteresis. In addition to effectively predicting dead blocks in the Last-Level Cache (LLC) by observing program phase behaviors, the replacement component also gives feedback to the prefetching component to help decide on the optimal fill level for prefetches. Meanwhile, the prefetching component feeds confidence information about each individual prefetch to the LLC replacement component. A low confidence prefetch is less likely to interfere with the contents of the LLC, and as confidence in that prefetch increases, its position within the LLC replacement stack is solidified, and it eventually is brought into the L2 cache, close to where it will be used in the processor core.

Third, we observe that the host machine in virtualized system operates under different memory pressure regimes, as the memory demand from guest Virtual Machines (VMs) changes dynamically at runtime. Adapting to this runtime system state is critical to reduce the performance cost of VM memory management. We propose a novel dynamic memory management policy called Memory Pressure Aware (MPA) ballooning. MPA ballooning dynamically speculates and allocates memory resources to each VM based on the current memory pressure regime. Moreover, MPA ballooning proactively reacts and adapts to sudden changes in memory demand from guest VMs. MPA ballooning requires neither additional hardware support, nor incurs extra minor page faults in its memory pressure estimation.

DEDICATION

To my loving creator, without his love and protection, I can do nothing.

ACKNOWLEDGEMENTS

First of all, I deeply appreciate my advisor, Paul V. Gratz, for his endless advice, insight, and support. He guided me to approach a technical problem from various angles, encouraged me to voyage the unexplored research area, and gave me numerous opportunities to work with greatest minds in both academia and industry. My research would not have been possible without his vision and professional experiences. I would also like to express my gratitude to my advisory committee members: Daniel A. Jiménez, A. L. Narasimha Reddy, and Jean-Francois Chamberland for their constructive feedback on my research, proposal, and this final dissertation.

I also thank to my colleagues at Computer Architecture, Memory Systems and Interconnection Networks (CAMSIN) and Texas Architecture and Compiler Optimization (TACO) research groups. Luke McHale, Eric Garfinkle, Gino Chacon, and Elvira Teran, it was my great pleasure to work with you all. I hope you will all continue the tradition of Whisky Club without having any loss of productivity. Special thanks to Chris Wilkerson, Alaa Alameldeen, Zeshan Chishti, and Seth Pugsley at Intel Labs. The collaboration with Intel researchers allowed me to learn the practical perspective of microarchitecture design.

Finally, I would like to thank my family who has mentally, spiritually, and financially supported me during my Ph.D. program. I am very fortunate to have this lovely family in my life.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Paul V. Gratz, Professor A. L. Narasimha Reddy, and Professor Jean-Francois Chamberland of the Department of Electrical and Computer Engineering and Professor Daniel A. Jiménez of the Department of Computer Science and Engineering.

Chapter II and III were collaborated with Elvira Teran of the Department of Computer Science and Engineering, and Chris Wilkerson, Seth Pugsley, and Zeshan Chishti of Intel Labs. Chapter IV was collaborated with Viacheslav Fedorov of NXP when he was a graduate student of the Department of Electrical and Computer Engineering.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by the National Science Foundation (NSF) through grants CCF-1320074 and I/UCRC-1439722, and a generous support by Intel Corporation.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iv

ACKNOWLEDGEMENTS v

CONTRIBUTORS AND FUNDING SOURCES vi

TABLE OF CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES xiii

CHAPTER I INTRODUCTION 1

I.1 Memory Wall 1
I.1.1 Data Prefetching 3
I.1.2 Cache Replacement Policy 4
I.1.3 Memory Management in Virtualized System 7
I.2 Dissertation Statement 7
I.3 Dissertation Organization 8

CHAPTER II CONFIDENCE-BASED MEMORY ACCESS PREDICTION . . 9

II.1 Introduction 9
II.2 Motivation and Prior Work 11
II.2.1 Speculating complex memory access patterns 11
II.2.2 Adapting Aggressiveness 12
II.2.3 Prefetching and Page Boundaries 13
II.2.4 Other Prior Prefetchers 14
II.3 Design 15
II.3.1 Design Overview 15
II.3.2 Learning Memory Access Patterns 18
II.3.3 Path Confidence-based Prefetching 20
II.3.4 Page Boundary Learning 22
II.3.5 Prefetch Filter 24

II.4 Evaluation . . . 26
II.4.1 Methodology . . . 26
II.4.2 Single Core Performance . . . 27
II.4.2.1 Prefetching Coverage and Accuracy . . . 29
II.4.2.2 Average Lookahead Depth . . . 31
II.4.2.3 Contribution to Performance Improvement . . . 32
II.4.3 Multi-programmed Mix Performance . . . 33
II.4.4 Sensitivity Study . . . 35
II.5 Summary . . . 37

CHAPTER III HOLISTIC MULTI-LEVEL CACHE MANAGEMENT . . . 38

III.1 Introduction . . . 38
III.2 Motivation . . . 40
III.2.1 Why do we need a holistic cache management? . . . 40
III.2.2 Why is a PC-based policy insufficient? . . . 42
III.2.3 Impact of Compiler Optimizations . . . 43
III.2.4 The PC can be replaced . . . 44
III.3 Design . . . 48
III.3.1 KPC-P: Confidence-based Prefeching . . . 49
III.3.1.1 KPC-P Overview . . . 49
III.3.1.2 KPC-P Training . . . 50
III.3.1.3 KPC-P Prefetching . . . 51
III.3.2 KPC-R: Global Hysteresis Replacement . . . 53
III.3.2.1 KPC-R Overview . . . 53
III.3.2.2 KPC-R Training . . . 54
III.3.2.3 KPC-R Placement/Replacement . . . 56
III.4 Evaluation . . . 58
III.4.1 Methodology . . . 58
III.4.2 Performance . . . 59
III.4.3 Analysis . . . 62
III.5 Summary . . . 66

CHAPTER IV DYNAMIC MEMORY REALLOCATION IN VIRTUALIZED SYSTEM . . . 67

IV.1 Introduction . . . 67
IV.2 Design Motivation . . . 71
IV.2.1 Adaptation to System Conditions . . . 71
IV.2.2 Slow Reclamation and Reallocation . . . 73
IV.2.3 Working Set Estimation Overhead . . . 75
IV.3 Design . . . 77
IV.3.1 Adaptive Memory Cushion . . . 77

IV.3.2 Memory Reallocation Trigger 79
IV.3.3 Adaptive Hysteresis 81
IV.3.4 Implementation 82
IV.4 Evaluation 83
IV.4.1 Methodology 83
IV.4.2 Repeating Single Application 85
IV.4.3 Multiple Applications in Random Order 87
IV.4.4 Hypercall Overheads 88
IV.5 Summary 90

CHAPTER V CONCLUSION 91

REFERENCES 93

LIST OF FIGURES

FIGURE Page

I.1 Memory access latency in terms of 3.2GHz processor cycle 1

I.2 Impact of prefetching depth on a simple PC-based delta prefetcher 4

I.3 Performance of SHiP [17] and EAF [18] under data prefetcher 6

II.1 Complex memory access pattern. 11

II.2 A case of lookahead prefetching caused by an infinite loop in the prediction pattern table. 13

II.3 Overall SPP architecture 17

II.4 SPP table update operations. 19

II.5 Path confidence-based lookahead prefetching. 20

II.6 Learning delta patterns across page boundaries. 23

II.7 Prefetching Filter. 25

II.8 Single-core IPC speedup. 28

II.9 Prefetching coverage and useless prefetches. 30

II.10 Raw prefetch request breakdown. Both VLDP and BOP generate significant numbers of useless prefetches. 31

II.11 Average Lookahead Depth 32

II.12 Contribution of SPP components. Stacked graph represents accumulated speedup from each component 33

II.13 Normalized speedup for mixes of 4 workloads. 34

II.14 Sensitivity study. 36

III.1 LLC allocation breakdown with DA-AMPM prefetcher 41

III.2 Loop unrolling example 44
III.3 Global hysteresis quickly trains and adapts to program phases 45
III.4 Dead block prediction accuracy for PC and global hysteresis 46
III.5 Design overview of the KPC system 48
III.6 Update the signature and delta pattern 50
III.7 KPC-P training and prefetching 51
III.8 KPC-R global hysteresis update mechanism 54
III.9 KPC-R updates fill level threshold for KPC-P 55
III.10 Single core performance compared to DA-AMPM + LRU 59
III.11 4-core multiprogrammed workloads performance 60
III.12 Various combinations of prefetching and replacement algorithms 61
III.13 Prefetching coverage: DA-AMPM vs. KPC-P 62
III.14 Dynamic adaptation of fill level thrshold T_F 63
IV.1 Performance degradation of ballooning with Tmem 68
IV.2 False balloon target due to clean pages in *vips* from the PARSEC suite . . 72
IV.3 Slow response to memory allocation and deallocation 74
IV.4 Adaptive memory cushion. 78
IV.5 Improved response time with trigger 79
IV.6 Performance analysis on trigger threshold 80
IV.7 Create memory pressure by restricting MEM_{host} 84
IV.8 Performance analysis in 20% pressure 86
IV.9 Performance analysis in 40% pressure 86
IV.10 Performance analysis in 60% pressure 86
IV.11 Performance analysis with random ordered applications 88

IV.12 Number of hypercalls normalized to Tmem 89

LIST OF TABLES

TABLE Page

II.1 Simulator parameters. 27

II.2 SPP storage overhead. 35

III.1 KPC-R prediction table 56

III.2 KPC storage overhead 64

III.3 Storage overhead comparison 65

IV.1 Design overhead of MP aware ballooning 82

IV.2 Baseline configuration 83

IV.3 Randomized mixes 85

CHAPTER I

INTRODUCTION

I.1 Memory Wall

In computer architecture, the memory hierarchy represents multiple levels of memory devices optimized for different speeds, capacities, and manufacturing costs. For example, on-chip caches are designed with expensive SRAM cells that provide fast response time while traditional off-chip main memory (DRAM) is optimized for higher capacity with lower implementation cost. With an advent of future memory technologies, such as Phase Change Memory (PCM) or flash-based NVMs, the discrepancy between memory devices becomes wider than ever. As a result, in modern memory system architecture, we can observe a huge range of differences in terms of response time, capacity, and bandwidth. Figure I.1 shows different memory technologies ordered in memory access latency. As illustrated in Figure I.1, a gigantic gap, also known as "Memory Wall" [1], exists between processor and main memory speed. Similarly, an access to the storage devices is significantly slower than the memory access. Although future NVM technologies can reduce the number of disk accesses by exploiting higher memory capacity, NVM is still far slower than DRAM and its limited bandwidth is also a major performance bottleneck.

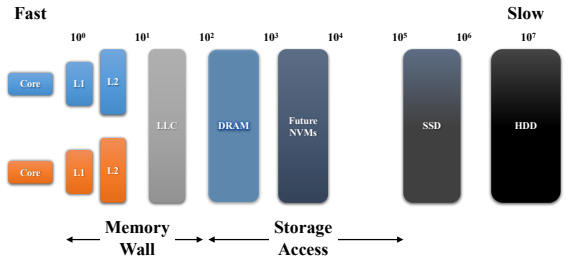


Figure I.1: Memory access latency in terms of 3.2GHz processor cycle

To mitigate the impact of slow memory accesses, major processor vendors have relied on a large multi-level cache hierarchy. With the end of Dennard scaling [2], however, growing cache size comes at an increasingly high cost in terms of power and energy consumption. As argued by Esmaeilzadeh *et al.* [3], energy consumption must be justified by increased performance to be practical as VLSI scaling continues; under this constraint the diminishing performance gains seen with increasing cache size become hard to justify relative to their energy cost.

To maximize a given cache capacity under the limited power budget, computer architects have focused on intelligent memory management techniques. In particular, advanced data prefetching and cache replacement algorithms have been deeply studied in modern microprocessor design. Ideally, if a core processor can accurately predict the future memory references, data blocks can be prefetched from off-chip memory ahead of its actual use or preserved for the next reference until the block does not exhibit any temporal locality in a given time period. In doing so, the overall system performance can be dramatically improved by serving most memory requests at the on-chip cache level without paying expensive main memory access latency. The important question is how to design an accurate and powerful speculation algorithm with minimal hardware complexity. An efficient memory management becomes more critical in large server clusters. For example, in virtualized environments, a guest machine does not know the available memory of the host system which prohibits efficient memory reallocation across multiple Virtual Machines (VMs). In this case, system architects need to adjust the total global memory pressure of system as well as local memory pressure in a guest machine. Since the communication between guest and host machines requires expensive system calls, it is also crucial to minimize the software communication overhead.

In the following sections, we will introduce basic backgrounds of these memory management techniques and states the contribution of this dissertation.

I.1.1 Data Prefetching

Prefetching is a well-studied technique which can provide an efficient means to improve the performance of modern microprocessors. The aim of prefetching is to proactively fetch useful cache blocks from further down in the memory hierarchy, ahead of their first demand reference. In essence, prefetching hardware speculates on the spatial and temporal locality of memory references, based on past program behavior. In some earlier proposed prefetching techniques, the prefetching opportunity is limited to waiting until a cache miss occurs, and then prefetching either a set of blocks sequentially following the current miss [4], a set of blocks following a stride pattern with respect to the current miss [5], or a set of blocks spatially around the miss [6, 7]. More recent prefetchers attempt to predict complex, irregular access patterns [8, 9, 10, 11, 12]. While these methods show significant benefit, because they are inherently reactive, the depth of their speculation is limited, which can lead to untimely prefetches.

Increasing prefetch depth is one way to speculate deeper. For example, if a processor has a simple next-line prefetcher that prefetches a (+1) delta ahead of the current demand cache block, we can build a more aggressive next *d* line prefetcher that prefetches $d*$(+1) deltas ahead of the current miss (*e.g.,* $+1_1$, $+1_2$, ..., $+1_d$). To generalize, a stride prefetcher whose delta distance (stride) is *N* and whose prefetching depth is *d* can be represented by (N_1, N_2, ..., N_d). Naively increasing the prefetching depth, however, does not always improve overall system performance, because often the predicted reference stream and the actual reference stream will eventually diverge. Figure I.2 shows the effect of prefetching depth on two different SPEC CPU 2006 benchmarks using a simple Program Counter (PC) based delta prefetcher [13]. As the prefetching depth *d* grows, the PC delta prefetcher brings more cache blocks that are *dN* distance away from base address. In this figure, *bwaves* benefits from deeper prefetching until $d = 7$, because its memory access pattern

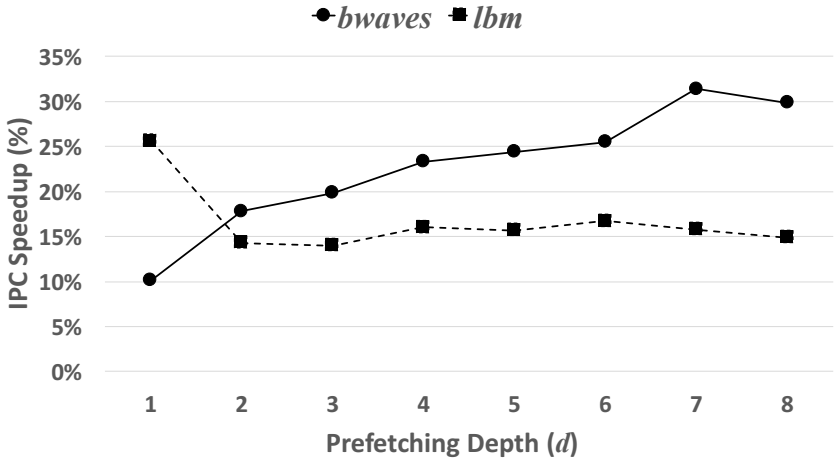


Figure I.2: Impact of prefetching depth on a simple PC-based delta prefetcher

is a predictable series of (+1) or (-1) deltas, up to seven steps ahead of the current demand access. After that point, the performance benefit drops as further speculative requests serve only to consume memory bandwidth, without contributing additional cache hits. On the other hand, *lbm* suffers from performance degradation as the prefetching depth grows. Since *lbm* has a variety of memory access patterns [11], deeper prefetching with a simple delta predictor wastes bandwidth, and pollutes the cache. While deeper speculation is useful for *bwaves*, *lbm* shows the greatest benefit from a speculative depth of only $d = 1$. Thus, as Figure I.2 illustrates, achieving high performance across many workloads requires adapting speculation relative to its prefetching accuracy. The adaptive controll mechanism is one of main motivations for path confidence-based prefetching and more details will be discussed in the later section.

I.1.2 Cache Replacement Policy

The Last-Level Cache (LLC) is a large on-chip structure with significant power consumption. In many cases, however, most of the LLC blocks are replaced without any reuse

during their life time in the cache. Khan et al. [14] showed that, on average, 86% of cache blocks in a 2MB LLC is dead and do not exhibit further reuse behavior. To improve the efficiency of LLC, advanced replacement algorithms [14, 15, 16, 17, 18] have been intensively discussed over the last ten years and showed superior performance than traditional LRU-managed LLC.

Still, without coordination between cache management and speculation techniques at different levels in the cache hierarchy, schemes such as data prefetching and replacement often work at cross-purposes. Previous work suggests advanced prefetching algorithms [19, 6, 20, 10, 21, 22, 23, 24] to reduce the gap between processor speed and memory latency. Most data prefetchers are trained by private L1 or L2 cache accesses to make timely prefetches far ahead of demand requests. Often, however, the appropriate placement within the shared LLC for these prefetched blocks is unclear. A sophisticated cache replacement policy is the right tool to solve this problem. Previous work [15, 16, 25, 26, 14, 17, 18, 27] shows a substantial gain can be achieved by placing blocks predicted to be dead [26] at the vulnerable position in the LRU stack. However, with data prefetches, the incremental benefit of replacement policy often becomes marginal or sometimes even negative [28, 29, 30].

In particular, replacement policies which use the Program Counter (PC) of missing load to predict reuse [25, 26, 14, 17, 27] experience substantial interference from prefetched blocks, which by definition do not carry demand fetch load PC values. Figure I.3 compares the IPC speedup of a top performing, PC-based replacement policy (SHiP [17]), and a recently proposed non-PC-based replacement policy (EAF [18]) when a high performance data prefetcher, DA-AMPM [29] is being used. In this figure, the performance is normalized to DA-AMPM with the baseline LRU replacement policy. Although SHiP typically shows better performance than EAF when running without prefetching, we see that here EAF outperforms SHiP across most applications. This is largely because PCs are

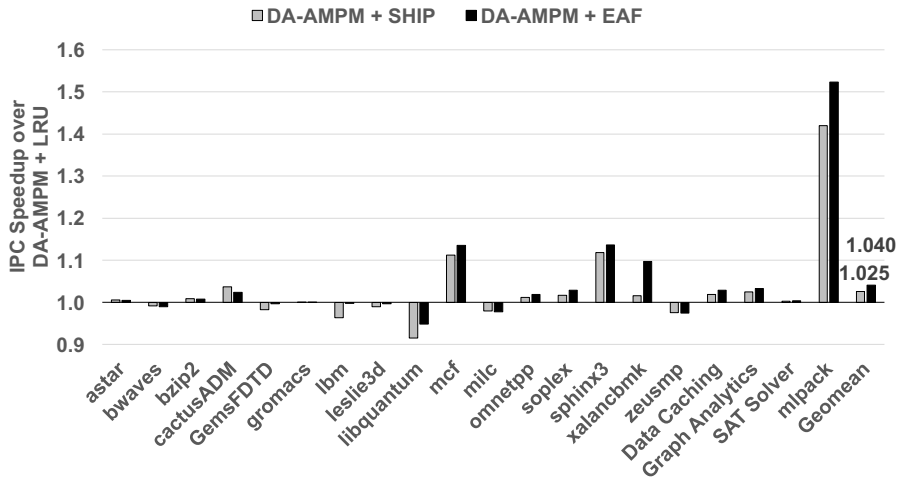


Figure I.3: Performance of SHiP [17] and EAF [18] under data prefetcher

simply not available for prefetches, forcing SHiP to use a static prediction (always dead or always live) for prefetched blocks. On the other hand, EAF tracks the physical addresses of recently evicted blocks in a bloom filter to make a dead block prediction. When there is a cache miss and the missing block is found in the victim filter, that block is inserted with higher priority. The baseline assumption is that if a block with high reuse is prematurely evicted from the cache, it will be accessed soon after eviction [18]. Thus, EAF can make a per-block-based prediction for both demand and prefetch that yields higher performance than SHiP without relying on PCs.

To alleviate the harmful interference between prefetching and replacement policy, several works [28, 29, 30] propose to selectively prioritize the prefetch request over demand request or vice versa. While these approaches show some gains, there remains little integration between the techniques, leaving critical program behavior information known by the prefetcher out of the replacement/placement decision, thus leaving performance on the table. In Chapter III, we will discuss how a holistic cache management can reduce the prefetching interference and improve the overall efficiency of multi-level caches.

I.1.3 Memory Management in Virtualized System

With the advent of Virtual Private Server (VPS) vendors, such as Amazon Elastic Compute Cloud (EC2) and Rackspace Cloud, cloud computing has become a large and growing component of the computing market. To provide adequate service for clients, vendors often utilize hardware virtualization due to its strong server consolidation and high scalability characteristics. In virtualized environments, a hypervisor or virtual machine monitor (VMM) is responsible for managing virtualized hardware resources and the execution of guest virtual machines (VMs). The main memory size specified by each guest VM running on a given host, however, can quickly add up to impose a high cost in required host machine physical memory. Despite this high cost, VMs rarely fully utilize their entire virtualized main memory space, thus wasting this valuable resource. To deal with under-utilization of physical memory in virtualized environments, hypervisor memory-management techniques which enable physical memory *overcommitment* have been developed, however they often trade lower system memory requirement for a significant performance cost, or do not effectively respond and adapt to dynamically changing memory pressure. The goal of Chapter IV is to reduce the performance overheads of hypervisor memory management techniques by proposing a Memory Pressure Aware (MPA) ballooning which dramatically improves responsiveness and adaptivity of virtualized system.

I.2 Dissertation Statement

In this dissertation, we introduce three reference-driven speculation mechanisms that can precisely predict the future memory usage. First, we propose a novel confidence-based data prefetcher. This prefetcher leverages confidence value to control the aggressiveness of data prefetching and extends its coverage by selecting a prefetching path with the highest confidence. Each prefetching path is represented by a series of signatures which is a

compressed form of past memory references. Second, we integrate both data prefetching and LLC replacement algorithms into one unified solution. Finally, we propose memory pressure aware ballooning that dynamically adapts to the global memory pressure state and reallocates memory resource in virtualized environments. Each prediction mechanism is designed to have minimal hardware or software complexity so that it can be easily implemented with the existing infrastructure.

I.3 Dissertation Organization

In the remaining chapters, each speculation mechanism is first introduced with the main research motivations and then followed by detailed implementation. Chapter II discusses the path confidence-based data prefetcher. Chapter III extends the concept of path confidence and proposes a holistic multi-level cache management technique. Chapter IV introduces the global memory pressure of virtualized system and shows how the overall system memory can be efficiently managed across multiple virtual machines. Proposed designs are evaluated with a set of thorough experiments. In Chapter II and III, we use a microarchitectural simulator to demonstrate the performance impact of proposed techniques. In Chapter IV, we directly modify the Linux kernel and Xen Hypervisor [31] to measure the real time performance. Finally, Chapter V concludes this dissertation.

CHAPTER II

CONFIDENCE-BASED MEMORY ACCESS PREDICTION*

This chapter presents a confidence-based prefetching mechanism that alleviates the impact of "Memory Wall". We first introduce a background of data prefetching. Then, a detailed implementation is described in the design section. The evaluation section compares our confidence-based prefetcher with other state-of-the-art data prefetchers.

II.1 Introduction

To address both prefetching coverage and accuracy, prior work has adopted lookahead mechanisms [11, 21, 32]. These studies, however, suffer from high hardware complexity [21, 32], or do not implement adaptive throttling [11]. For example, B-Fetch [21] and Runahead [32] require either deep hooks into the core microarchitecture or a whole secondary core to prefetch accurately, making them impractical to implement in the lower levels of the cache. Meanwhile, recent techniques, such as VLDP [11], showed that a lookahead path can be built by memory access pattern without relying on core pipeline information. This algorithm, however, prefetches a static depth ahead of the demand fetch, without considering the prefetching confidence [11], either leading to missed opportunities when the accuracy is high or to inaccuracy in the face of divergent memory access patterns. Moreover, most hardware prefetchers work in the physical address space [6, 11, 12, 13], where the mapping between virtual and physical memory is not known. As a result, it is often difficult to predict patterns across 4KB physical page boundaries.

In this chapter, we propose a simple but powerful path confidence-based lookahead prefetcher, the Signature Path Prefetcher (SPP). The contributions of this work are:

*Reprinted with permission from "Path Confidence based Lookahead Prefetching" by J. Kim, S. H. Pugsley, P. V. Gratz, A. L. Narasimha Reddy, C. Wilkerson, and Z. Chishti 2016. Proceedings of the 2016 49th International Symposium on Microarchitecture, Copyright 2016 by IEEE

- We introduce a signature mechanism that stores memory access patterns in a compressed form and initiates the lookahead prefetching process. Up to four small deltas can be compressed in this 12-bit signature without aliasing. By correlating the signature with future likely delta patterns, SPP learns both simple and complicated memory access patterns quickly and accurately. The signature can be also used to detect the locality between two physical pages, and continue the same prefetching pattern off the end of one physical page and onto the next.

- We develop a path confidence-based prefetch throttling mechanism. As lookahead prefetching goes deeper, a series of signatures builds a signature path. Each signature path has a different confidence value based on its previous delta history, prefetching accuracy, and the depth of prefetching. The path confidence value is used to throttle prefetching depth dynamically in order to balance prefetch coverage with accuracy.

- Unlike prior lookahead based prefetchers [21, 32], SPP does not require deep hooks into the core microarchitecture and is purely based on the physical memory access stream.

We evaluate SPP with a combination of SPEC CPU 2006 and commercial workloads and find it achieves an average 27.2% performance improvement compared to a baseline without prefetching. Moreover, SPP outperforms recent, best of class, lookahead and non-lookahead prefetchers [10, 11, 12], including the winner of the most recent data prefetching competition, by 6.4% on average. The remaining sections are organized as follows. Section II.2 discusses the motivation for path confidence based prefetching. Section IV.3 describes the detailed hardware implementation of SPP. A detailed performance evaluation is presented in Section III.4.

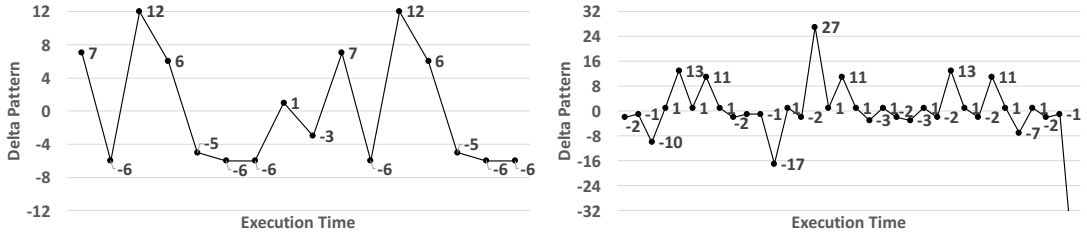
II.2 Motivation and Prior Work

In this section, we examine previously proposed prefetchers with an eye toward improvement. In particular, we note that complex data access patterns are difficult to predict without the use of recursive lookahead mechanisms, while existing lookahead-based prefetchers do not consider path confidence, instead prefetching to an arbitrary degree.

II.2.1 Speculating complex memory access patterns

An optimal prefetching algorithm should cover a wide range of memory access patterns. Simple stride prefetching techniques only detect sequences of addresses that differ by a constant value and fail to capture diverse delta patterns [33]. For example, Figure II.1 shows two examples of complex memory access patterns, taken from *GemsFDTD* and *mcf*, which cannot be captured by a simple prefetcher. Though both show complicated patterns, *GemsFDTD* (Figure II.1a) has a repeating sequence of strides (+7, -6, +12, +6, -5, -6, -6), that should be predictable assuming the prefetcher can store a long delta history. Concatenating such a long sequence in a simple pattern table, however, could result in huge storage overhead.

On the other hand, Figure II.1b shows that *mcf* has a random (though biased) access pattern that is difficult to predict. In this particular case, it is better to use a simple next-line



(a) *GemsFDTD*: Complex but predictable pattern (b) *mcf*: Complex and random patterns

Figure II.1: Complex memory access pattern.

prefetcher, because (+1) is the most commonly seen delta pattern. Offset based prefetchers such as the Best Offset prefetcher [12] and the Sandbox prefetcher [22] evaluate multiple offsets at run-time and issue prefetches with an offset that maximizes likelihood of use. These offset prefetchers do not, however, account for temporal ordering between delta patterns, and suffer from low accuracy on complex, yet predictable address patterns. In addition, if there are multiple offsets that are commonly observed during program execution, offset prefetchers take longer to train or fail to select the optimal offset. Lookahead prefetchers [11, 21] efficiently encode the relationship between accesses to yield future predictions, enabling further speculative lookahead accesses.

II.2.2 Adapting Aggressiveness

Lookahead prefetchers learn patterns by collecting histories of observed data access patterns, and correlating these with the next expected delta in the pattern. Figure II.2 shows an example lookahead prefetcher that recursively refers to a pattern table to generate future prefetches. In this example, the prefetcher indexes into the pattern table to find the next predicted delta for prefetching. Once this prefetch is issued, the prefetcher recursively uses that prediction to again index into the pattern table and generate further predictions. This recursion allows lookahead prefetchers [11, 21] to prefetch far ahead of the current program execution, and generate timely prefetches for as long as their predictions remain accurate.

In principle, the lookahead process can be repeated as long as the predicted pattern is found in the pattern table. As shown in Figure II.2, delta (+1) predicts the 3rd index (+3), and delta (+3) predicts the 1st index (+1), forming a loop. While the loop here may persist for many iterations, it is unlikely to persist forever, thus the true desired prefetching depth must be limited. To avoid over-prefetching, existing lookahead prefetchers [21, 11] globally and statically limit the depth to which lookahead is pursued ahead of the current de-

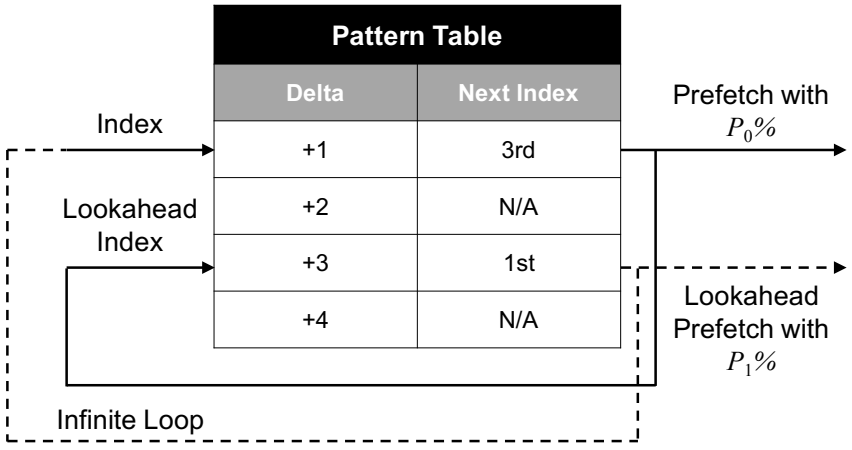


Figure II.2: A case of lookahead prefetching caused by an infinite loop in the prediction pattern table.

mand access stream. Unfortunately, it is often the case that the ideal prefetch depth varies from application to application, as shown in Figure I.1, and even varies between prefetch streams within the same application. Thus a per-prefetch stream throttling mechanism is critical to adapt prefetch aggressiveness to the stream's prediction confidence [20].

II.2.3 Prefetching and Page Boundaries

Virtual memory is a vital tool in the operation of modern computers, but it creates unique challenges when designing data prefetchers which work in the lower-levels of the cache. Two addresses which are contiguous in the virtual address space might be separated by great distances in the physical address space. By extension, patterns which are trivially easy to detect in the virtual address space might be nearly impossible to fully detect in the physical address space.

Prefetchers are often located alongside caches where they have no access to address translation hardware, and hence they do not have knowledge of the relationship between physical pages in the virtual address space. Thus, data access patterns which cross physical

page boundaries are difficult to exploit. Physical address prefetchers often try to learn intra-page or global patterns, applying these patterns to new pages that are encountered, or they discover simple patterns, like streams or strides, by considering each new page in a vacuum. This has the effect of either ignoring complex patterns altogether, as in the case of the Best Offset prefetcher [12], or requiring long, per-page, warmup times, as in Access Map Pattern Matching [10] and conventional stream prefetchers. All of these prefetchers must stop prefetching once a page boundary is reached, because it is impossible to know the physical mapping of the next page in the virtual address space. Although challenging to implement, an effective prefetcher should be able to seamlessly continue complex patterns detected in one page as they cross page boundaries, without having re-detect the pattern from scratch in the new page.

II.2.4 Other Prior Prefetchers

Somogyi *et al.* proposed Spatial Memory Streaming (SMS) [6] which leverages the correlation between memory request instruction addresses (PC) and access patterns spatial near the current memory request. This purely spatial pattern ignores the temporal ordering between future demand accesses. Later efforts extended this approach to detect the temporal order between delta patterns [8, 9]. While these prefetchers achieve good performance, they require megabytes of hardware state storage, which is orders of magnitude more than the other prefetchers considered in this paper.

II.3 Design

To address the deficiencies outlined in Section II.2, we propose the Signature Path Prefetcher (SPP), a novel, low-overhead and accurate lookahead prefetcher.

II.3.1 Design Overview

SPP uses a speculative mechanism we refer to as "lookahead" to increases its prefetching degree and improve timeliness. Once a delta prediction has been made by SPP, and a prefetch request has been issued, the accumulated history used to make the prediction(*i.e.,* the signature) is speculatively extended to include the predicted delta, thereby generating a new lookahead signature (the mechanism is discussed in detail in Section II.3.2). This signature can then be used to make a new delta prediction, which leads to producing yet another signature. Thus, we use predicted deltas, with no confirmation of their accuracy, to recursively speculate down a "*signature path.*" These speculative signatures resemble techniques used in branch prediction to deeply speculate beyond unresolved branches. Relying on this mechanism, SPP can continue to speculate much further than the most recently confirmed delta. In fact, with this mechanism in place, the challenge shifts from generating new delta predictions to deciding when to stop.

From the discussion in Section II.2.2 of Figure II.2, we can conclude that lookahead prefetches will tend to be less accurate than initial prefetches. In general, as the number of speculative deltas increases, our confidence in the prefetch requests should decrease as the product of the confidence of each speculative prefetch along the path. To illustrate this, assume the probability of the first $delta_0$ prediction being accurate is p_0 and the probability of the second $delta_1$ is p_1. During speculative lookahead, the prefetch resulting from $delta_1$ has a probability of being accurate of $p_0 * p_1$. This is because the path used to select $delta_1$, contains $delta_0$ which itself is only accurate with probability p_0. In fact, the probability that the delta predicted at any lookahead depth of *d* will produce

a useful prefetch is a product of the probabilities of all prior speculative deltas ($\prod p = p_0 * p_1 * p_2 * ... * p_d$). We label this path probability P_d as the product of the probability of all constituent deltas to depth d such that $P_d = \prod p$. Further, the probability of any path of depth *d* can be expressed as a function of the probability of the most recent delta and the probability of the previous path, as shown in Equation II.1:

$$P_d = p_d \cdot P_{d-1} \tag{II.1}$$

P_d can be compared against a given confidence threshold to adaptively determine when prefetching for this particular lookahead stream should be stopped. Further, this confidence, P_d, can also be used to determine which cache level to insert prefetched lines, with more accurate/confident prefetches sent to a higher cache level and less accurate/confident prefetches sent to a lower cache level.

To gracefully handle the transition between physical pages, SPP shares learning across page boundaries. As discussed in Section II.2.3, SPP operates in the physical address space, without any access to address translation. Thus, the spatial relationships between physical pages are largely unknown and assumed to be random. Despite this, SPP leverages learning from one physical page to make timely predictions in other pages. SPP does this in two ways, first, while delta history signatures are maintained on a per-physical page basis, as will be described in Section II.3.2, those signatures index into a global table for predicting deltas, which is shared by all pages. Second, as described in Section II.3.4, when the first demand access is made to a new physical page, the delta signature patterns from a previous physical page whose delta predictions crossed a page boundary can be inherited and used to bootstrap predictions in this new page. This gives SPP the advantage of not requiring long, per-page warmup periods to start prefetching complex patterns, which leads to higher prefetch coverage.

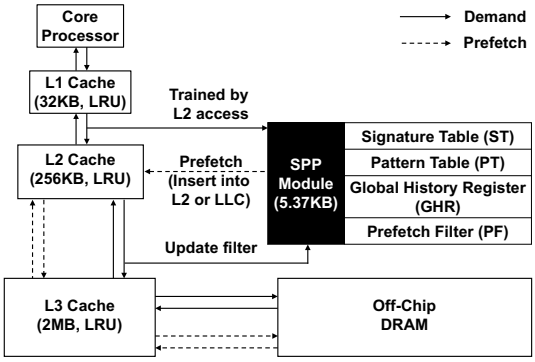


Figure II.3: Overall SPP architecture

The high level design of the SPP engine is illustrated in Figure II.3. The SPP module consists of three main structures (Signature Table, Pattern Table, and Prefetch Filter) and a small Global History Register (GHR), which is used for cross-page boundary bootstrapped learning. SPP is trained by L2 cache accesses, and can fill prefetch requests into either the L2 or LLC (depending on the confidence of the predicted prefetch). The Signature Table (ST) tracks the 256 most recently used pages, and stores the history of previously seen delta access patterns in each page as a compressed 12-bit signature. The Pattern Table (PT) is indexed by the history signatures generated by the ST and stores predicted delta patterns. The PT also estimates the path confidence that a given delta pattern will yield a useful prefetch. If the delta generated by the PT is found to have sufficient confidence (above a configured threshold), then it is passed as a prefetch candidate to the Prefetch Filter (PF), which checks for redundant prefetches. If the predicted delta crosses a 4KB physical page boundary, SPP does not issue the prefetch but instead redirects the request to the GHR for page boundary learning. In the remainder of this section we describe each stage of the SPP in detail.

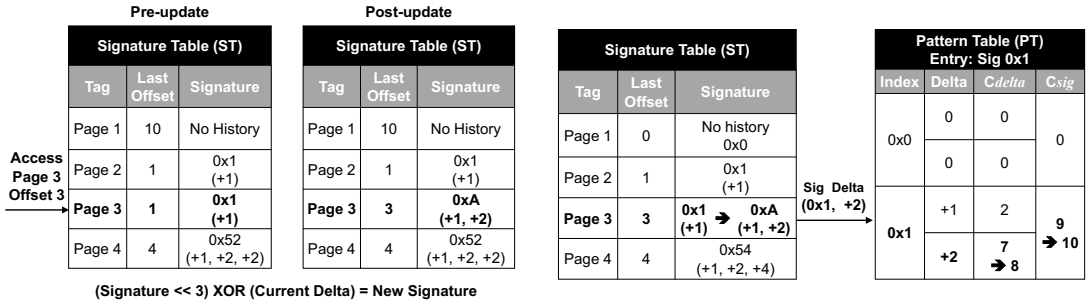
II.3.2 Learning Memory Access Patterns

The ST, shown in Figure III.6, is designed to capture memory access patterns within 4KB physical pages, and to compress the previous deltas in that page into a 12-bit history signature. The ST tracks the 256 most recently accessed pages, and stores the last block accessed in each of those pages in order to calculate the delta signature of the current memory access, which is then used to access and update the PT. Whenever there is an L2 cache access, its physical address is passed to the ST to find a matching entry for the corresponding physical page. Figure III.6 shows an example of accessing the ST with a physical page number of 3 and block offset 3 from the beginning of the page. In this case, the ST finds a matching entry for this page and is able to read a stored signature 0x1. This signature is a compressed representation of a previous access pattern to that page, which was generated via a series of XORs and shifts as shown in Equation II.2. In this case, the signature 0x1 represents a single previous delta access to Page 3, which was (+1).

$$\begin{aligned} &\textit{New Signature} \\ =&(\textit{Old Signature} << \textit{3-Bit}) \text{ XOR } (\textit{Delta}) \end{aligned} \tag{II.2}$$

Since the ST stores the last block offset 1 accessed in Page 3, we know that the current delta in Page 3 is (3 - 1) = (+2). This delta is non-speculative, because it is based on a demand request to the L2. Therefore, we can infer that a given set of accesses (signature 0x1 in this instance) will lead to a delta of (+2). Figure II.4b shows how this correlation is used to update the delta pattern in the PT.

The probability of each delta occurring is approximated using a per-delta confidence value C_d, which is derived from counters stored in the PT. Since a matching delta (+2) that corresponds to 0x1 is found in the PT, the corresponding C_{delta} counter increases by one.



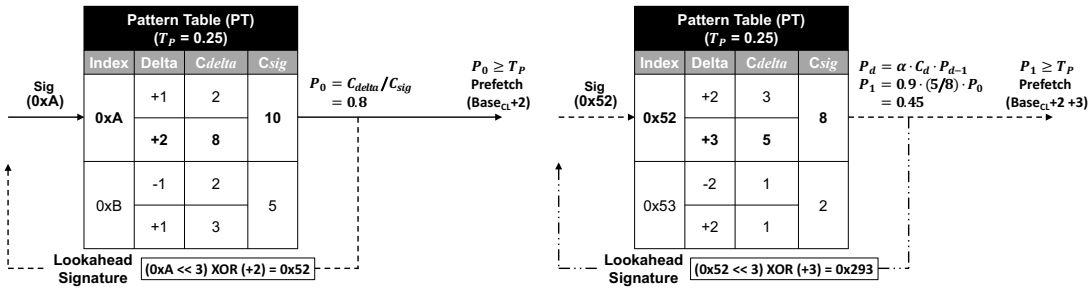
(a) Update signature (b) Update delta pattern

Figure II.4: SPP table update operations.

In order to estimate the prefetch accuracy probability for each delta, we also maintain an separate counter C_{sig}, which tracks the total occurrences of the signature itself. If either C_{delta} or C_{sig} saturates, all counters associated with that signature are right shifted by 1. In doing so, SPP is able to continue updating its counters with the most recent information without ever completely losing all previously collected history. If the PT contains no matching delta, we simply replace the existing delta with the lowest C_{delta} value.

Unlike the ST, whose entries correspond to individual physical pages, each PT entry is shared globally by all pages. If Page A and Page B, for example, share the same access pattern, they will generate the same signature, which indexes to the same entry in the PT, and updates the same delta pattern in the PT. Sharing patterns between pages in the PT minimizes SPP training time as well as the number of entries needed to store predictions. Each entry in the PT can hold up to four different deltas so that multiple different deltas can be prefetched by a single signature. Each of the deltas in a PT entry can be prefetched if its corresponding probability ($C_d = C_{delta}/C_{sig}$) is above the given prefetching threshold T_P.

After updating the PT, the ST is also updated with a new signature based on the current delta (+2). Equation II.2 shows how the SPP generates a new history signature. The old



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

Figure II.5: Path confidence-based lookahead prefetching.

signature 0x1 is left-shifted 3-bits and XORed with the current delta (+2). In this way, a 12-bit signature can represent the last four memory accesses in Page 3. Note that we refer to this signature as being "compressed" because deltas greater than 7 have the potential to overlap and cause aliasing with existing history. At this point, the new signature (0x1<<3) XOR (+2) = 0xA represents the current access pattern, (+1,+2), in Page 3. Assuming 4KB pages with 64B cache lines, all possible deltas fall in the range of (-63) to (+63). We use a 7-bit sign+magnitude representation for both positive and negative deltas. Thus, negative and positive deltas produce different signatures, pointing to different entries in the pattern table, and ultimately different prefetch targets.

II.3.3 Path Confidence-based Prefetching

After updating the ST, as in Figure III.6, SPP accesses the PT so that it can predict the next delta following signature 0xA. As shown in Figure II.5a, the path confidence P_d is calculated for deltas associated with signature 0xA. The initial path confidence P_0 is simply set by (C_{delta} / C_{sig}) since there is no prior path confidence value. In this example, (+2) delta has $P_0 = 0.8$ which is greater than the prefetching threshold T_P. Therefore, the PT adds the delta (+2) to the current cache line's base address and issues a prefetch request.

In addition, the PT initiates the lookahead process by building a speculative *lookahead signature*. As shown in Figure II.5a, the lookahead signature 0x52 is generated from 0xA and the predicted delta (+2) using Equation II.2. While there could be multiple candidates for prefetching, SPP only generates a single lookahead signature, choosing the candidate with the highest confidence. The lookahead signature is used to index the PT again so that SPP can search for further prefetch and lookahead candidates down the signature path. If the lookahead signature 0x52 finds more prefetch candidates in the PT (Figure II.5b), the process will be repeated and prefetch requests will be issued. The lookahead mechanism is used recursively until the signature path confidence P_d falls below the prefetching threshold T_P. Note, a separate, greater confidence threshold T_f determines which level of the cache a given prefetch will be fill into, either the L2 if P_d is greater than T_f or the LLC if P_d is less than T_f.

SPP also uses a global accuracy scaling factor α based on the observed global prefetching accuracy to further throttle down or increase the aggressiveness of the lookahead process. Thus, Equation II.1 is modified to include α according to Equation II.3:

$$P_d = \alpha \cdot C_d \cdot P_{d-1} \quad (\alpha < 1) \tag{II.3}$$

If the prefetching accuracy is generally high across all prefetches, α will decrease the path confidence P_d very slowly, allowing deeper lookahead prefetching. On the other hand, if global prefetching accuracy is low, α will quickly throttle down lookahead prefetching. The prefetching accuracy is tracked by the PF described in section II.3.5. The global α and the local, per-delta C_d work together to provide a throttling mechanism which globally adapts to the general prefetching confidence of a given program phase, while simultaneously favoring some signature paths over others based on their relative confidence.

Note that for simplicity, in the discussion above we describe path confidence calcula-

tions in the context of floating point numbers. However, in a real implementation (and in our simulator), we use 7-bit fixed point numbers to represent path confidence values between 0∼100, and perform multiplication and division on those fixed point numbers. In addition, since C_{delta} and C_{sig} are 4-bit saturating counters, we can use a simple 16x16=256 entry lookup table that stores all possible division results, which allows us to completely remove the expensive divider modules. Moreover, the extra computational latency can be hidden, because SPP can calculate the path confidence in the background while the L2 cache is waiting for the DRAM to service demand misses.

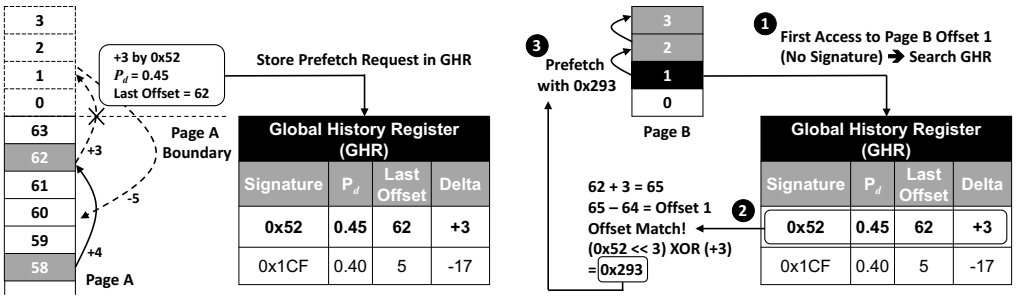
In addition to throttling when P_d falls below the prefetch threshold T_p, SPP also stops prefetching if there are not enough L2 read queue resources. Reserving L2 read queue entries is desirable because even accurate, useful prefetches can take resources away from even more performance-critical demand misses from the L1 cache. Therefore, SPP does not issue prefetches when the number of empty L2 read queue entry becomes less than the number of L1 MSHRs. To summarize, SPP stops prefetching if the prefetcher observes one of following conditions:

1. Low path confidence P_d.

2. Too few L2 read queue resources.

II.3.4 Page Boundary Learning

One of the challenges for history-based prefetchers like SPP is the loss of history information that occurs during transitions to new physical pages. Pages that are adjacent in the virtual address space may not be adjacent in the physical address space. As a result, patterns that are very easy to learn and follow in the virtual address space may be very hard to detect and predict when they are broken up by a physical page transition. Previously proposed history-based techniques [6, 11] address this by making predictions using the



(a) GHR stores prefetch request that crosses physical page boundary

(b) Immediately start prefetching for new page

Figure II.6: Learning delta patterns across page boundaries.

first offset in a page, or with very short delta histories. Although these predictions may increase coverage, they do so at the expense of accuracy because they are made with little or no information about what happened in the previously accessed page.

SPP addresses this problem with a novel mechanism that allows histories to be maintained and tracked across physical page transitions. It does this by augmenting the per-page ST with a global history that is updated when the prefetcher makes a prediction off the end of a page, and is checked against when accessing new pages for the first time.

Figure II.6 shows how SPP connects a signature path across physical page boundaries without the need to relearn any delta history patterns, or do any other warmup in the new page. As shown in Figure II.6a, when there is a prefetch request that goes beyond the current Page A, a conventional streaming prefetcher must stop prefetching, because it is impossible for the prefetcher to predict the next physical page number. However, this boundary-crossing prediction can still be useful when the next page is accessed for the first time, and we find that the page offset of that first access matches the out-of-bounds offset previously predicted by SPP.

In order to track this behavior, the boundary crossing prediction is stored in a small 8-entry GHR. The GHR stores the current signature, path confidence, last offset, and delta

used for the out-of-page prefetch request, which is everything necessary to bootstrap SPP prefetching in a new page. If we access a new page that is not currently tracked (*i.e.,* a miss in the ST), SPP searches for a GHR entry whose last offset and delta match the current offset value in the new page. Figure II.6b shows that Page B is accessed with an initial offset of 1. SPP checks if any GHR entry's last offset and delta value match the current offset of 1. In this case, the signature 0x52 has the last offset and delta whose sum (62 + 3 = 65) matches the offset of 1 in Page B, since there are only 64 64B blocks in 4KB physical page. We can now predict that Page B will produce a delta pattern that is a continuation of the signature 0x52.

Since the pattern predicted by signature 0x52 is now continuing in a new page, we need to connect the signature 0x52 with delta (+3) by generating a new signature. Using the same Equation II.2, we generate a new signature 0x293 for Page B that can begin prefetching immediately, without needing to learn any additional delta history. The new signature 0x293 is entered into the ST for future use by Page B. Thus, SPP does not suffer from long, per-page warmup periods, and it can prefetch complex patterns in new physical pages sooner, resulting in higher prefetch coverage. Unlike the Global History Buffer [34], which records all observed delta patterns, the GHR only stores delta patterns that cross page boundaries. Note that SPP does not stop looking even further ahead for more prefetch candidates after coming to the end of a page and updating the GHR. As shown in Figure II.6a, if the (+3) delta, which lands in Page B, is predicted to be followed by a (-5) delta, which comes back to Page A, then SPP can still exploit this behavior and prefetch an offset of 60 in Page A.

II.3.5 Prefetch Filter

The main objectives of the PF, shown in Figure II.7, are to decrease redundant prefetch requests, and to track prefetching accuracy. The PF is a direct-mapped filter that records

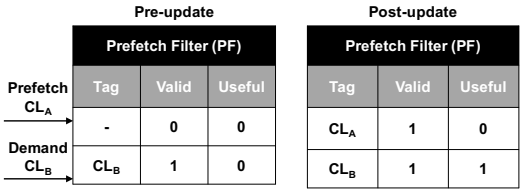


Figure II.7: Prefetching Filter.

prefetched cache lines. SPP always checks the PF first, before it issues prefetches. If the PF already contains a cache line, this means that line has already been prefetched, and SPP drops the redundant prefetch request. Entries in the filter get cleared by resetting a *Valid* bit when the corresponding cache line is evicted from the L2 cache.

Due to collisions, a filter entry may already be occupied by another prefetched cache line. In this case, SPP simply replaces the old cache line, stores the current prefetch request in the filter, and issues the current prefetch. Note that this simple replacement policy might erase cache lines from the filter before they get evicted from the L2 cache, which could lead to re-prefetching, but we find in practice that this happens very infrequently.

By adding a *Useful* bit to each filter entry, the PF can also approximate prefetching accuracy. SPP has two global counters, one which tracks the total number of prefetch requests (C_{total}), and the other which tracks the number of useful prefetches (C_{useful}). The C_{total} counter increases whenever SPP issues a prefetch that is not dropped by the filter. Useful prefetches are detected by actual L2 cache demand requests hitting in the PF, which increments the C_{useful} counter. To avoid increasing C_{useful} more than once per useful prefetched line, we set a used bit in the PF entry which keeps it from being double counted. The global prefetching accuracy tracked by this filter is used for α in Equation II.3 to throttle the path confidence value.

II.4 Evaluation

In this section, we evaluate the SPP prefetch engine. We first present the evaluation methodology, followed by single core and multi-core performance. Finally, we present a sensitivity study of SPP's design parameters.

II.4.1 Methodology

We evaluate SPP using the ChampSim simulator, which is an updated version of the simulation infrastructure used in the 2nd Data Prefetching Championship (DPC-2) [35]. We model 1-4 out-of-order cores, whose parameters can be found in Table II.1. ChampSim is a trace-based simulator, and we collect SimPoint [36] traces from 18 memory intensive SPEC CPU2006 [37] applications. We also collect single thread traces from 3 server workloads (Data Caching, Graph Analytics, SAT Solver) from CloudSuite [38]. Since our SimPoint methodology does not work with the server workloads, we instead collect the server workload traces after fast-forwarding at least 30B instructions to pass through the benchmark's initialization phase. For performance evaluation, we warm up each core for 200M instructions and collect results over an additional 1B instructions.

Single core simulations use a single DRAM channel. In multi-core simulations, all cores share a single L3 cache and main memory system, with two DRAM channels. Instruction caching effects are not modeled in this simulation infrastructure. In ChampSim, all prefetching actions are initiated by an L2 access, but prefetches can be directed to fill in either the L2 or the L3 cache. Our simulation infrastructure uses a 4KB page size when mapping virtual to physical addresses. In ChampSim, virtual to physical page mappings are arbitrarily randomized. All of the prefetchers we evaluate in this work were designed to operate strictly in the physical address space with no knowledge of the relationship between physical and virtual address spaces.

We compare SPP against three top performing recently proposed prefetching algo-

Core Parameters	1-4 Cores, 3.2 GHz 256 entry ROB, 4-wide 64 entry scheduler 64 entry load buffer
Private L1 Dcache	32KB, 8-way, 4 cycles 8 MSHRs, LRU
Private L2 Cache	256KB, 8-way, 8 cycles 16 MSHRs, LRU, Non-inclusive
Shared L3 (LLC)	2MB/core, 16-way, 12 cycles, LRU, Non-inclusive
Main Memory	1-2 64-bit channels 2 ranks/channel, 8 banks/rank 1600 MT/s

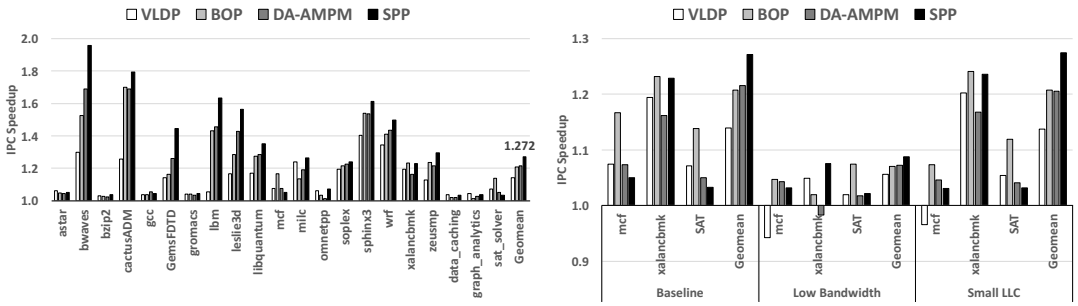
Table II.1: Simulator parameters.

rithms: the Variable Length Delta Prefetcher (VLDP) [11], the Best Offset Prefetcher (BOP) [12], and the DRAM-Aware Access Map Pattern Matching (DA-AMPM) [39] prefetcher[†] . We use the original code for each of these prefetchers submitted to DPC-2. In each case, their design parameters have been re-tuned to attain their highest performance in ChampSim running these traces. SPP does not use any feature of ChampSim that was not also available to the designers of the other evaluated prefetchers. SPP's threshold configurations were empirically derived. The prefetching threshold T_P = 0.25 and fill level threshold T_F = 0.9 were found to provide good performance improvement and accuracy, and they are used throughout these results except where otherwise noted.

II.4.2 Single Core Performance

Figure II.8 shows the IPC speedup of all four evaluated prefetchers over a no-prefetching baseline. Overall, **SPP** outperforms or matches all other prefetchers on nearly every benchmark. On average, **SPP** achieves a 27.2% geometric mean speedup, which is 6.4% and 5.6% more than **BOP** and **DA-AMPM** respectively. Also, **SPP** outperforms **VLDP** (a recent lookahead prefetcher) by 13.2%. **SPP** shows particularly significant improvement

[†]Note: DA-AMPM is an extended version of AMPM [10] which accounts for DRAM row buffer locality.



(a) IPC speedup versus no prefetching. (b) IPC speedup for Low BW and small LLC.

Figure II.8: Single-core IPC speedup.

in benchmarks that have complex access patterns. For example, both *GemsFDTD* and *lbm* show substantial performance improvement with **SPP**. Further, **SPP** shows remarkable performance gains in *bwaves, cactusADM, leslie3d,* and *libquantum*. These applications are highly predictable by SPP, and gain significantly from SPP's ability to adaptively look ahead very deeply.

As Ferdman *et al.* [38] noted in a prior study, data prefetching for large scale workloads is not as effective as prefetching for general purpose workloads. Performance with **SPP** only improves by 2∼4% for this class of benchmark. However, **SPP** and **VLDP** show the best performance for Data Caching and Graph Analytics, because both prefetchers are able to capture the complex memory access patterns found in those workloads. **BOP** achieves a 13.8% performance gain in *SAT solver* by aggressively prefetching on every L2 cache access. The impact of aggressive prefetching by **BOP** will be discussed further in the next section. We also modeled and compared the performance of the SMS [6] prefetcher originally designed for server workloads, however, the overall performance of SMS on both SPEC and CloudSuite was less than that of DA-AMPM. To simplify the performance analysis, we only include the results from DA-AMPM.

Figure II.8b shows the performance improvement under different memory resource

constraints. The baseline configuration used for Figure II.8a has 12.8GB/s of DRAM bandwidth and 2MB LLC. The low bandwidth test is configured with only 3.2GB/s memory bandwidth, and the small LLC configuration has only 512KB of last level cache. Along with a geomean across all benchmarks, Figure II.8b also shows performance for three benchmarks where **SPP** has lower performance than other prefetchers. As shown in Figure II.8b, due to aggressive prefetching, **VLDP** and **BOP** show similar or worse performance improvement for *mcf* and *xalandbmk* under the low bandwidth configuration. Meanwhile, **BOP** always shows the best performance for *mcf* and *SAT solver* regardless of resource constraints. We find that these benchmarks have random access patterns (*i.e.,* pointer chasing) that cannot be easily captured in the physical address space. Therefore, offset-based prefetchers [12, 22] work better than **SPP** for these, because they are trained by individual offset occurrence frequency, and do not rely on longer delta patterns repeating. However, the benefit of aggressive offset prefetching without pattern matching can be nullified by resource constraints when multiple workloads fight for limited shared LLC and DRAM bandwidth [40]. The performance degradation of **BOP** with *mcf* and *SAT solver* is discussed in the multi-core analysis section.

II.4.2.1 Prefetching Coverage and Accuracy

The substantial performance benefit of **SPP** versus the other prefetchers shown in Figure II.8 is a direct result of **SPP's** prefetching accuracy and coverage. Figure II.9 shows the prefetching coverage for each benchmark. In this figure, each prefetcher is indicated by its first letter (*e.g.*, "V" for **VLDP**, "D" for **DA-AMPM**, etc.). Prefetching coverage is measured by the number of useful prefetches divided by the number of cache misses without prefetching. Because prefetching can be useful in different ways, we further break down the useful prefetches into four different categories. **Useful** represents the portion of prefetched cache lines that are filled into the cache before their first demand access, elimi-

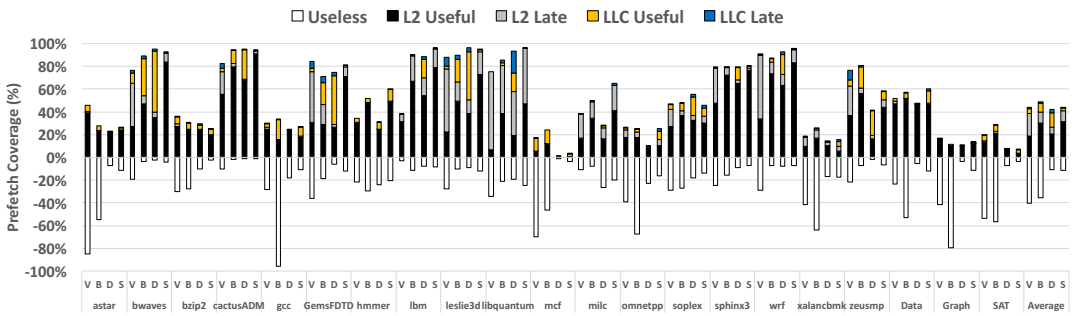


Figure II.9: Prefetching coverage and useless prefetches.

nating all cache miss penalty. On the other hand, **Late** represents the portion of prefetched cache lines that were issued to DRAM but were not filled before their first demand access. Thus, **Late** prefetches have the effect of accelerating the demand fetch compared to a regular cache miss. **Useful** and **Late** are measured for both the L2 cache and the LLC. We also plot the fraction of useless prefetches on the bottom of the figure, showing the effect of over-aggressive, low-confidence prefetching. On average, **BOP** shows the highest prefetching coverage, however, it also generates a large number of useless prefetches (mostly to the L2 cache), which consume additional DRAM bandwidth and energy. Although SPP has slightly lower coverage compared to BOP, it has the largest number of L2 **Useful** prefetches due to its highly accurate dynamic path confidence-based throttling.

Because prefetching coverage alone does not show the aggressiveness of each prefetcher, we also plot the raw number of prefetch requests in Figure II.10. As discussed in Section II.4.2 and shown in Figure II.8a, **BOP** and **VLDP** outperform **SPP** in three workloads (*mcf*, *xalancbmk* and *SAT solver*) in a single-core environment. Figure II.10 shows that for these workloads, **BOP** and **VLDP** generate a huge number of useless prefetches to achieve this performance advantage, aggressively consuming DRAM bandwidth and LLC capacity. In a single core environment, this over-prefetching does not negatively impact

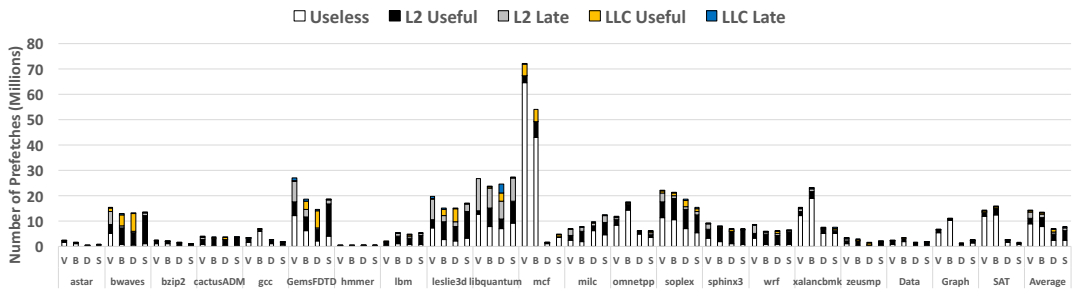


Figure II.10: Raw prefetch request breakdown. Both VLDP and BOP generate significant numbers of useless prefetches.

performance, however as we will show, performance does degrade when the LLC and DRAM bandwidth are shared by multiple cores.

II.4.2.2 Average Lookahead Depth

Because the length of memory access patterns is different in each application, the optimal lookahead prefetching depth also varies. Figure II.11 shows the diversity of lookahead depths SPP uses across the 21 benchmarks we examined. For example, *libquantum* is well known to have very stable and long delta patterns. Figure II.11 confirms that SPP uses a very deep lookahead depth for *libquantum*. Meanwhile, *milc* also uses a deep lookahead depth, but this case is very different from *libquantum*. We find that many delta patterns in *milc* do not fit within a single 4KB page. In these cases, SPP will predict that the access pattern will leave the current page, and then, several deltas later, return to the current page, where prefetching can continue. SPP will continue looking further ahead until its confidence falls below the threshold. On the other hand, **SPP** does not prefetch deeply on *xalancbmk*, because this benchmark is not amenable to prefetching, due to a low degree of spatial and temporal locality in each page. In this case, **SPP** detects that prefetching is inaccurate by using the PF, and lowers the global scaling factor α, which serves to limit the lookahead depth. The average lookahead depth of **SPP** across all traces is 6.9.

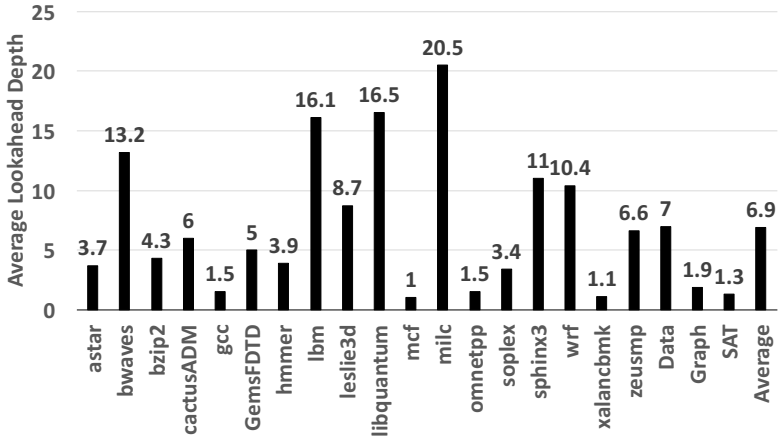


Figure II.11: Average Lookahead Depth

II.4.2.3 Contribution to Performance Improvement

Figure II.12 shows the relative contribution of lookahead and page boundary learning (via the GHR), compared to a basic version of **SPP** without either of these features. To highlight the impact of each feature, we select two benchmarks (*libquantum* and *GemsFDTD*), and break down the performance benefit. A basic **SPP** algorithm without lookahead or the GHR shows a performance improvement of 11.8% on *libquantum* and 14.0% on *GemsFDTD*. Adding lookahead prefetching achieves a significant additional speedup of 21.4% for *libquantum*, because the benchmark has simple memory access patterns that can be accurately covered with very deep lookahead prefetching. On the top of that, page boundary learning provides little benefit for *libquantum*, because each page takes very little time to train, even withouth the GHR. For *GemsFDTD*, lookahead prefetching improves performance by 25.9%. However, page boundary learning provides a substantial boost of 4.5% to *GemsFDTD* because many delta patterns do not fit inside a single 4KB page. In this application, a common delta pattern is (+30) followed by (+1), thus many of the deltas cross page boundaries. The GHR structure captures this behavior

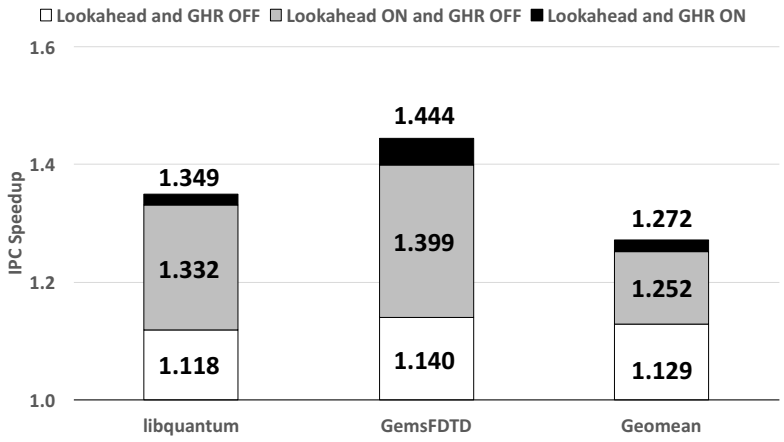


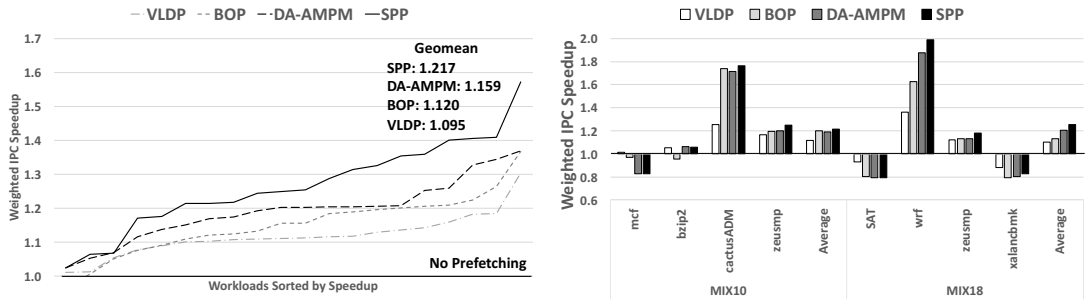
Figure II.12: Contribution of SPP components. Stacked graph represents accumulated speedup from each component

in *GemsFDTD* and provides a substantial performance improvement. On average, basic **SPP**, without lookahead provides a 12.9% speedup; adding lookahead prefetching provides 12.3% more speedup, and finally adding the page boundary learning gains another 2.0% improvement.

II.4.3 Multi-programmed Mix Performance

To show results for a multi-core system, we generate 20 multi-programmed mixes consisting of traces from different benchmarks, and assign each trace to a different core. The mixes are randomly generated in order to fairly represent the characteristics of multi-programmed workloads. Figure II.13a shows the performance improvement of four-workload mixes, measured by normalized weighted speedup. The graph is sorted by speedup order. Out of the 20 random mixes, **SPP** achieves the best performance on 19 mixes. For the remaining mix, **DA-AMPM** beats **SPP** by less than 0.5%. On average, **SPP** achieves a 21.7% speedup compared to the baseline.

Note that the **BOP** prefetcher, which nearly ties for second place on single core bench-



(a) Workloads sorted by normalized weighted speedup

(b) Performance alaysis for MIX10 and MIX18.

Figure II.13: Normalized speedup for mixes of 4 workloads.

marks, shows particularly poor performance in multicore systems. As shown in Figure II.9, although **BOP** has good coverage, it also fetches a high number of useless lines. In a single core system this does not hurt performance, because the LLC is not overcommitted. In a multicore system, however, the LLC pressure is greater, which leads to performance loss. Figure II.13b shows two examples of multi-programmed mixes that suffer from **BOP**'s aggressive prefetching. Each mix contains benchmarks in which **BOP** outperforms **SPP** in the single core environment. In MIX10, **VLDP** and **BOP** show the least performance degradation on *mcf*. However, this performance in *mcf* comes from aggressive prefetching, which prevents other workloads from getting better performance. In particular, **BOP** shows performance degradation on *bzip2* while the other prefetchers show performance improvement. In MIX18, all prefetchers gain performance in *wrf* and *zeusmp* at the cost of degrading *SAT solver* and *xalancbmk*. Similarly, **BOP**'s advantage is lower than **SPP**'s due to its aggressive prefetching on *SAT solver* and *xalancbmk*. Note that in single core experiments, **BOP** showed performance improvement in *xalancbmk* and *SAT solver*. However, when resources are shared among multiple cores, **BOP**'s aggressive prefetching hurts overall system performance.

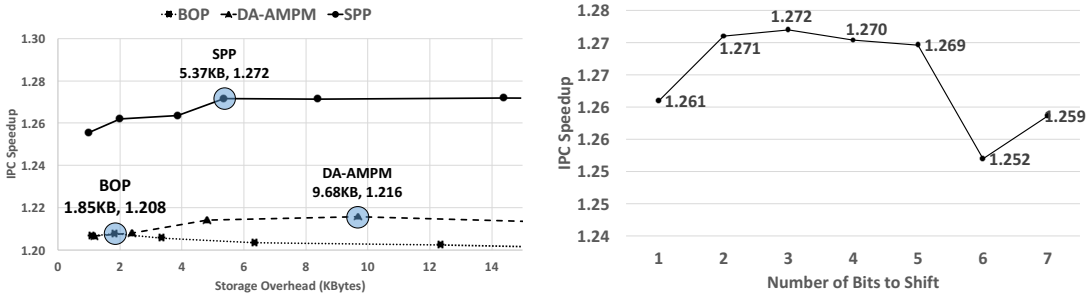
II.4.4 Sensitivity Study

The total storage used by **SPP** in the preceding experiments is 5.37KB, with the storage requirement for each individual component shown in Table III.2. As the table shows, **SPP**'s largest component is the PT (3KB). This table has multiple delta predictions and counters for each tracked signature. Figure II.14a shows a performance sensitivity analysis between **BOP**, **DA-AMPM**, and **SPP** when scaling storage size. Each point represents the storage configuration used for the performance evaluation. As expected, the overall performance of **SPP** and **DA-AMPM** does not increase with greater storage capacity. Generally, **SPP** always outperforms the other prefetchers at a given storage budget. Note that, counterintuitively, **BOP** does not benefit from greater storage, because it takes a longer time to find the best offset value when more cache lines are in its recent request table [12].

SPP uses a 12-bit history signature built by a series of 3-bit shifts and XORs. This shifting and XORing represents a form of lossy information compression. The more bits

Structure	Entry	Component	Storage
Signature Table	256	Valid (1 bit) Tag (16 bit) Last offset (6 bit) Signature (12 bit) LRU (6 bit)	11008 bits
Pattern Table	512	C_{sig} (4 bit) C_{delta} (4*4 bit) Delta (4*7 bit)	24576 bits
Prefetch Filter	1024	Valid (1 bit) Tag (6 bit) Useful (1 bit)	8192 bits
Global History Register	8	Signature (12 bit) Confidence (8 bit) Last offset (6 bit) Delta (7 bit)	264 bits
Accuracy Counter	1 1	C_{total} (10 bit) C_{useful} (10 bit)	10 bits 10 bits
11008 + 24576 + 8192 + 264 + 20 = 44060 bits ≈ 5.37 KB			

Table II.2: SPP storage overhead.



(a) SPP storage sensitivity.

(b) Performance sensitivity with respect to the number of bits to shift.

Figure II.14: Sensitivity study.

that are shifted, the less compression, and vice-versa. Here we examine the performance impact of this compression. Figure II.14b shows the performance sensitivity to the number of shifted bits. Since each delta pattern within a 4KB page can be represented with 7 bits (-63 $\sim$ 63), there is no value in shifting more than 7 bits. The figure shows that a 12-bit signature prefers 3-bit shifting. The goal of the history signature is to track as many useful deltas as possible within a small 12-bit signature. This is best achieved by giving small deltas, which are the most common, all the precision they need, while still keeping some information from larger deltas. By shifting 3-bits prior to XORing, SPP compresses four deltas into 12-bits. Deltas from 0-7 are represented at full precision. Larger deltas (>7) cause some aliasing, but still contribute information to the final signature.

II.5 Summary

Signature Path Prefetching offers compelling solutions for three major prefetching challenges. First, it is able to learn and prefetch complex data access patterns by using a compressed history signature. Second, it is able to detect when a data access pattern crosses a page boundary, and quickly resume prefetching on the new page. Third, it is able to balance aggressive prefetching with accuracy by using path confidence. SPP is able to do all this without using the program counter or other core registers, and while operating strictly in the physical address space. SPP improves performance versus a no-prefetching baseline by 27.2%, and improves performance by 6.4% versus the next best competing technique.

CHAPTER III

HOLISTIC MULTI-LEVEL CACHE MANAGEMENT*

This chapter presents Kill the Program Counter (KPC), a holistic cache memory management technique that covers both data prefetching and cache replacement policy. First, we analyze why Program Counter (PC) cannot be a perfect tool to speculate future program behavior with the presence of data prefetcher. Based on our analysis, we propose a simple global hysteresis mechanism that can accurately predict the program behavior in the LLC. The proposed technique is further extended to provide constructive feedback to the data prefetcher living in the private cache.

III.1 Introduction

Due to the combined pressures of increasing application working sets [38, 41], the persistence of the Memory Wall [1], and the breakdown of Dennard scaling [3], processor memory system hierarchies have continued to grow in complexity, size, and performance criticality. In recent architectures, caches consume as much as 40% of the total die area and 20% of energy consumption on chip [42]. With so much of the available on-die resources invested in the cache hierarchy, an efficient, high performance design requires intelligent cache management techniques. While many cache management and speculation techniques such as alternate replacement policies [15, 16, 25, 43, 44], dead-block/hit prediction [26, 14, 17, 18, 27], and prefetching techniques [19, 6, 20, 10, 21, 22, 23, 24] have been extensively explored, many of these are piecemeal, one-off solutions that often interact poorly when implemented together and typically only address one level of

*Reprinted with permission from "Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy" by J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson 2017. Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, Copyright 2017 by ACM

the memory-system hierarchy. There has been little work exploring the interactions between these policies across multiple levels of the memory hierarchy and examining the information needed across boundaries in the system from software to the core, to the last level cache. This study proposes a holistic, speculative, multi-level cache management system that effectively reconstructs program behavior in the processor memory hierarchy to prefetch and manage placement of data across the cache hierarchy.

In this work, we present a novel cache management mechanism called Kill-the-PC (KPC) that integrates data prefetching and replacement policy across multiple levels of the cache hierarchy into a single system. KPC consists of two main components. First, we develop a prefetcher (KPC-P) that produces a proxy of future use distance based on its prediction confidence. KPC-P is an extended version of SPP [23] and tightly integrated with the LLC management technique. Second, we propose a replacement policy (KPC-R) that quickly adapts to the dynamic program phase using two small global counters. Each counter is exclusively updated by demand or prefetch so that KPC-R can predict useless cache blocks for both memory requests.

Additionally, KPC-R and KPC-P are integrated to share information. For example KPC-P learns from KPC-R to dynamically adjust a threshold for the prefetching fill level. KPC-R monitors a sample of prefetched blocks in the LLC to check if they could have been timely prefetches at the L2 cache. If there are enough timely prefetches detected by KPC-R, the fill level threshold becomes lower allowing prefetches go all the way up to the L2 cache. On the other hand, if the L2 cache is being polluted by a low fill level threshold, KPC-P automatically increases the threshold. Further, when KPC-P sends prefetch requests from the L2 to hit in the LLC with a given confidence, that confidence is used to update placement information withing the LLC's replacement stack. Critically, neither component depends on load PCs, eliminating the hardware complexity of PC propagation through the entire on-chip cache hierarchy.

III.2 Motivation

In this section, we discuss the need for holistic cache management and explain why the PC is an inadequate input feature for holistic cache design, especially under the effect of prefetching.

III.2.1 Why do we need a holistic cache management?

Previous studies [20, 28, 30] show that a large fraction of prefetches are dead in the LLC. Figure III.1 analyzes the types of allocations within the LLC when the DA-AMPM prefetcher is in use. More than 40% of LLC allocations are filled by prefetching and approximately 90% of these prefetches are useless, *i.e.*, they will have no accesses in the LLC (they are also pulled into the L2 and all hits to them occur there). Ideally, an intelligent cache replacement policy should detect these dead prefetches and evict them as soon as possible from the LLC. However, as we noted in Figure I.3, both PC- and non-PC-based replacement algorithms often yield negative impact when they are combined with prefetching. If we further break down the usage of prefetched blocks, it is even harder for a replacement policy to predict correct reuse behavior by itself. Some prefetched blocks are constantly reused by demand (white) while some blocks are only hit by another prefetch request from the L2 (grey).

In order to minimize the interference from prefetching, Wu *et al.* propose PACMan [28], a prefetch-aware cache management policy. PACMan dedicates a few sets of the LLC to each of the three competing policies that treat demand and prefetch requests differently and uses the policy that shows the lowest number of cache misses. Competition between three different policies, however, increases the overhead of set dueling [15], especially in a multicore environment. Similarly, Seshadri *et al.* propose ICP [30]. ICP is also designed as a comprehensive mechanism to mitigate prefetching interference. ICP simply demotes a prefetched block to the lowest priority on a demand hit based on the observa-

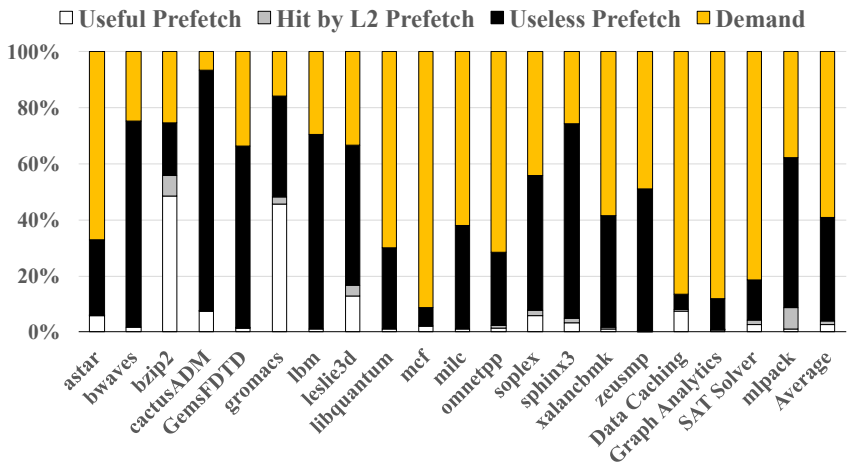


Figure III.1: LLC allocation breakdown with DA-AMPM prefetcher

tion that most prefetches are dead after their first hit. To address prefetcher-caused cache pollution, it also uses a variation of EAF [18] to track prefetching accuracy and inserts only an accurate prefetch to the higher priority position in the LRU stack. ICP assumes, however, that all prefetches are inserted only into the LLC, which restricts the maximum benefit of prefetching. Additionally, with sophisticated, high-performance data prefetchers [21, 29, 23, 22, 24], demoting a prefetched block on the first hit actually degrades the overall performance. In fact, we found that EAF [18] shows better performance than ICP with lower hardware complexity and storage overhead when it is combined with the DA-AMPM prefetcher. Critically, both PACMan and ICP consider the data prefetcher as an independent component that disturbs the LLC replacement policy and attempts to isolate that disturbance. Neither technique attempts to leverage information from the replacement policy to produce better prefetching algorithm.

Without a holistic approach that identifies how prefetched blocks are used in the L2 and LLC, we cannot optimize the efficiency of the precious on-chip cache resource. A Unified Memory Optimizing (UMO) architecture [29] is the most recent work to attempt holistic

cache design. UMO's main idea is to design a data prefetcher that increases the DRAM row buffer locality (DA-AMPM) and a replacement policy that refers to the data prefetcher for better prediction accuracy. However, UMO needs to access the L2 prefetcher on every LLC access since its replacement policy depends on the status map of DA-AMPM. Further, its prefetching algorithm is still separated from the LLC replacement policy and operates as a stand alone module. More importantly, UMO assigns equal priority to a stream of prefetches whose future use distance can be different from each other. In fact, we find that PACMan and EAF achieve higher performance than UMO when they are combined with DA-AMPM, which necessitates a better approach for holistic cache design.

III.2.2 Why is a PC-based policy insufficient?

One way to implement a holistic approach is to use a PC from the core pipeline for both prefetching and replacement policy. Passing PCs throughout the load-store queue and the all levels of cache hierarchy, however, requires extra logic, wire, and energy consumption. Additionally, there is a significant organizational cost of extra communication between front-end, mid-pipe, cache design, and verification teams as new interfaces are defined, implemented, and tested. When time-to-market is considered, incorporating the PC into prefetching and replacement may be considered too costly by industrial microarchitects. Moreover, modern data prefetchers [22, 24, 29, 11] do not associate prefetches with a particular PC. Thus, when the LLC allocates a cache line brought by a prefetch request, there is no PC value that can be used for reuse distance prediction. Even for demand requests, the PC does not always correlate with reuse behavior.

The baseline assumption of PC-based replacement algorithms is that a given memory instruction will exhibit certain memory use behavior over the program execution, regardless of which particular data location that instruction references. This is not always true, however, since a single load or store instruction might show variable cache line reuse be-

havior. For instance, if there is a load instruction located in a nested loop, data brought by that load may or may not be reused depending on the result of prior branches. In this scenario, using a single PC cannot provide a robust prediction. Instead of using a single instruction address, it is possible to accumulate a history of PCs. Although Lai *et al.* [26] introduced and Liu *et al.* [45] later refine a dead block predictor that collects a trace of PCs, using multiple PCs does not improve accuracy in the LLC since most memory accesses are filtered by upper-level caches, causing many important PCs to be missed [14].

III.2.3 Impact of Compiler Optimizations

Compiler optimizations, such as loop unrolling, also affect the performance of PC-based replacement algorithms. Contrary to the prior case where a single PC exhibits multiple reuse behaviors, loop unrolling generates multiple PCs that often show the same reuse behavior, necessitating a larger prediction table to correlate PCs and reuse. Figure III.2a shows a typical example of a loop structure that loads data from an array and reuses it in the same loop iteration. Without loop unrolling, the original loop code will be repeated n times. As a result, a single load instruction (PC Y) will be executed n times and the resulting data (X) will each be reused once. In other words, PC Y is responsible for n data reuses. Prior PC-based replacement algorithms are designed to observe these n instances of data reuse and update a prediction table by increasing the reuse counter associated with PC Y.

With compiler loop unrolling, the actual correlation between PCs and data reuse transforms as shown in Figure III.2b. The compiler generates n consecutive load instructions and places them ahead of the reuse function. In doing so, we can reduce the number of branch instructions and achieve more memory level parallelism. Unlike the original code, the unrolled loop contains n load PCs and each PC is associated with a single reuse rather than the n reuses without unrolling. To capture this reuse correlation without any conflicts

```
for (i=0; i<n; i++) {
  X = load(arr[i]);   // PC Y loads arr[i]
  Z = reuse(X);       // PC Y is reused
}
```

(a) Original loop

```
X1 = load(arr[0]);       // PC Y1 loads arr[0]
...
Xn = load(arr[n-1]);     // PC Yn loads arr[n-1]

Z1 = reuse(X1);          // PC Y1 is reused once
...
Zn = reuse(Xn);          // PC Yn is reused once
```

(b) Unrolled loop

Figure III.2: Loop unrolling example

between PCs, the PC-based replacement algorithm requires at least n entries in its prediction table. Furthermore, training n entries will take n times more iterations through the code as a non-unrolled version. Since loop unrolling is a common optimization technique, PC-based algorithms will suffer from hardware overheads and less prediction accuracy.

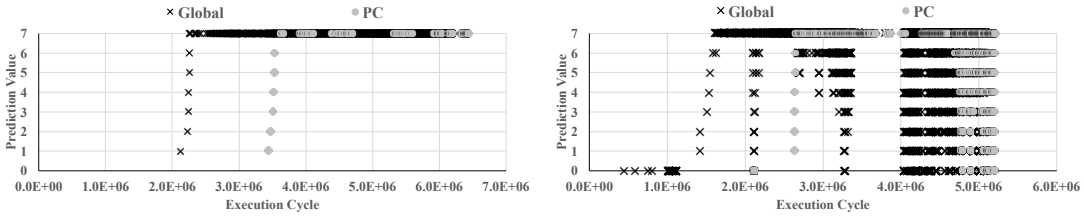
Note that, in the unrolled example, not every PC will be used to access the predictor on every unrolled iteration, since an initial demand read will load multiple words causing subsequent iterations to hit in the L1. Because alignment of data structures is not required to be on LLC block boundaries, however, the particular PC that causes a miss will vary from one entry to the unrolled loop to the next.

III.2.4 The PC can be replaced

Instead of extracting PCs from the front-end of core pipeline, we propose using a simple global hysteresis mechanism that quickly adapts to the dynamic program phase and provides similar or higher prediction accuracy than PC-based prediction. Figure III.3 shows how the prediction counter value changes over the program execution for PC-based

prediction and global hysteresis prediction. For PC-based prediction, we profile the most frequently used PC in two SPEC CPU 2006 benchmarks and track a prediction counter value correlated to that PC. In this experiment, we use the prediction mechanism proposed in SHiP [17]. If a cache block is evicted without being reused, the PC that allocated this block increases its prediction counter value. When the counter reaches the maximum value of 7, cache blocks brought by this PC are considered to be dead. Otherwise, if a cache block is hit, the corresponding PC prediction counter decreases. For the global hysteresis experiment, we use a single 3-bit counter that is updated by every LLC hit and miss.

Figure III.3a shows the training of the prediction counter in cactusADM. This benchmark has a long streaming access pattern in which all references are effectively dead on arrival. Both prediction techniques eventually saturate their prediction counters and predict most incoming cache blocks to be dead. Some cache blocks with more temporal locality may be preserved to be reused in the LLC. In the figure, we see that the global hysteresis approach learns faster than the PC-based because it is updated on every LLC access while only a subset of references touches any given entry in the PC-based. Figure III.3b shows that the prediction counter changes frequently in sphinx3 because the working set size is slightly larger than the size of LLC and there are some LRU friendly blocks. Still, the



(a) cactusADM training time

(b) sphinx3 training time

Figure III.3: Global hysteresis quickly trains and adapts to program phases

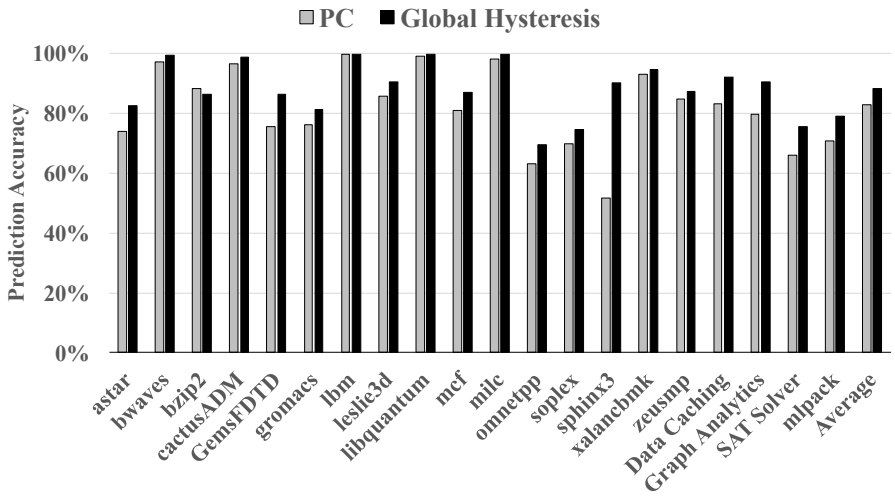


Figure III.4: Dead block prediction accuracy for PC and global hysteresis

global hysteresis adapts to the program phase much faster than the PC-based prediction for this workload.

The advantage of fast learning and dynamic adaptation results in better prediction accuracy. Figure III.4 shows the prediction accuracy for PC and global hysteresis predictions. As we expected, cactusADM shows similar prediction accuracy for both PC and global hysteresis. For this workload even the slower training of PC-based is sufficient because the streaming access pattern does not change. Alternately, the global hysteresis prediction produces much higher prediction accuracy for sphinx3. As described above, the global mechanism more quickly adapts to the program behavior changes seen in this application.

Why does the simple global hysteresis prediction work? There are two main reasons behind this. First, the global hysteresis makes a dead block prediction only when the counter is saturated. A single cache hit can change the direction of prediction. Thus, there is a low risk of discarding useful cache blocks. Second, data structures with similar temporal locality tend to be accessed in a similar time frame. Typically, when a programmer

writes an application, he or she works at the data structure level (*i.e.,* map, list, queue) rather than hardware cache block level. As a result, when a function call accesses a data structure, the cache blocks in that specific structure tend to be accessed at the same time. Thus, a small global counter can track the reuse behavior of the LLC without complex hardware. As a result, a simple global counter can replace a set of logics and wires that deliver PCs from core pipeline to the cache memory hierarchy.

III.3 Design

Here we examine the design of our proposed holistic cache management algorithm, KPC. KPC has two primary components: KPC-P the prefetching component, and KPC-R the cache replacement algorithm. These components are co-designed and integrated to achieve high-performance through the reconstruction of program behavior in the cache hierarchy.

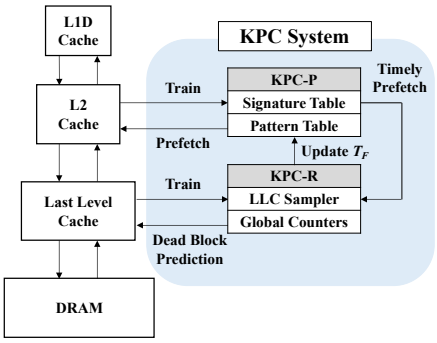


Figure III.5: Design overview of the KPC system

Figure III.5 provides a high level design overview of KPC. KPC-P is trained by L2 demand accesses and issues each prefetch with a calculated prediction confidence. The confidence value is used to control prefetch throttling, prefetching level within the cache hierarchy (*i.e.*, L2 or LLC), and prefetch promotion. A prefetch is issued only if its confidence is higher than a set prefetching threshold constant (T_P)[†]. Typically, a prefetch whose predicted use is far in the future is given a low confidence. KPC-R is trained by both prefetch and demand LLC accesses. A small fraction of LLC references are sampled to update the LLC sampler and predictor. KPC-R also dynamically updates the fill level

[†]We empirically determined 25% for this threshold was optimal.

threshold (T_F) based on feedback about prefetch timeliness from KPC-P. Thus, the entire KPC module provides a holistic cache management scheme.

III.3.1 KPC-P: Confidence-based Prefeching

Next we describe in detail the design and mechanisms of the KPC-P prefetching algorithm.

III.3.1.1 KPC-P Overview

KPC-P is designed to produce a per-prefetch confidence value that controls the aggressiveness of prefetching. Inspired by prior work on lookahead prefetching [21, 23, 11], KPC-P monitors the pattern of cache block accesses in a physical page, and then recursively prefetches future cache blocks in that page until its prediction confidence falls below a threshold, T_P. To achieve this goal, the Signature Table (ST), shown in Figure III.5, records a compressed history of past L1 cache block misses as a history of deltas (*i.e.*, differences) between consecutive memory addresses to the current physical page. This history, including the delta between the last reference in the current page and the current reference in the current page, is used as a signature to index into the Pattern Table (PT) to look up the predicted next delta in the current page. Once this prefetching prediction is made, the PT also generates a "lookahead" signature, corresponding to the predicted next reference in the current page. This lookahead signature is used to re-reference the PT and produce another predicted next delta, and in turn to produce another lookahead signature, and so on.

Figure II.4 illustrates the prefetching mechanism of KPC-P when an L2 access occurs to physical page 0xB with an offset of 2 in that page. In the next two subsections we describe the operation of KPC-P using this example.

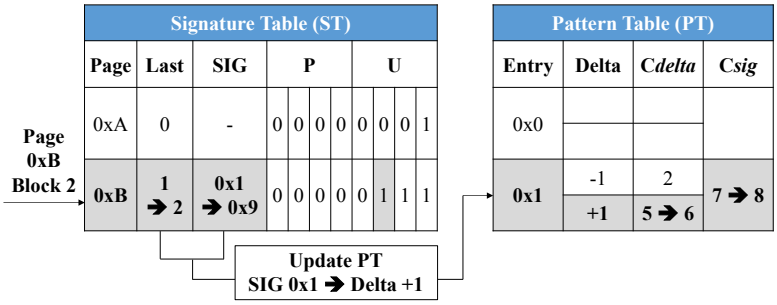


Figure III.6: Update the signature and delta pattern

III.3.1.2 KPC-P Training

When the L2 cache accesses a block of offset 2 in page 0xB, KPC-P begins by searching for a matching page entry in the ST. Figure III.6 shows that the page 0xB was recently accessed, is currently being tracked by the ST, and that the last block offset accessed in this page was 1. Further, we see that the most recent delta history signature in this page (SIG) is 0x1. Thus, the PT is indexed with the signature 0x1. We see that the PT currently has two valid deltas stored at index 0x1, -1 and +1. Because the current reference delta of +1 matches one of these, the +1 delta entry's occurrence counter (C_{delta}) is incremented, adding confidence to the prediction of a signature of 0x1 leading to a delta of +1. The signature occurrence counter (C_{sig}) is also incremented. We will discuss in Section III.3.1.3 how these two counters are used to estimate prefetch confidence. Each physical page has its own ST entry, but all pages contribute to building up predictions in a single PT, which is shared by all pages. This accelerates delta pattern learning times.

Based on the current reference, the last block offset and history signature in the ST must also be updated. A new history signature 0x9 is constructed by shifting the prior signature (0x1) to the left 3-bits and XORing it with the current delta (+1). This new value is stored as the new current signature. The last offset into the page (Last) is also

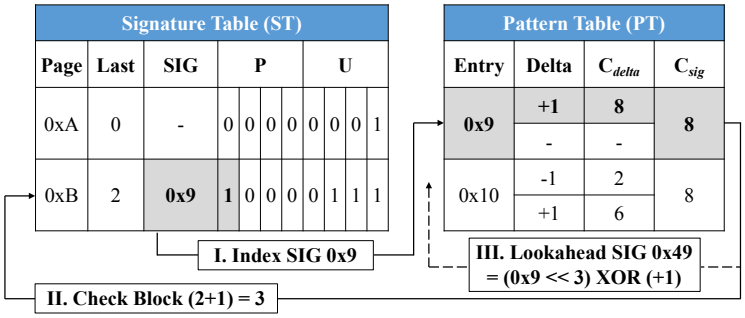


Figure III.7: KPC-P training and prefetching

updated to 2. The ST also maintains two bit-vectors to track the status of each cache block within each page. The prefetched (P), and used (U) vectors work together to ensure cache blocks are not redundantly prefetched, and enable the calculation of a general prefetching effectiveness metric, which is used to throttle future prefetches. The used bit is now set, which prevents block 2 from being prefetched again by any future predictions. Finally, we check to see if the prefetch bit corresponding to block offset 2 was previously set. If it was, then prefetching for that block is considered to be timely. The ST resets both prefetch and used bits when a block is evicted from the L2 cache.

III.3.1.3 KPC-P Prefetching

After updating both tables, KPC-P begins confidence-based prefetching, as illustrated in Figure III.7. First, (**I** on the figure) the PT is indexed by the updated signature 0x9. Any of the deltas in a PT entry can be prefetched, as long as their corresponding confidence, calculated as $C_0 = C_{delta}/C_{sig}$, is above the prefetching threshold T_P. In this example, 0x9 was seen 8 times and each time it was always followed by a +1 delta, giving a confidence of $8/8 = 1$ or 100%. Based on this 100% confidence, KPC-P will plan to prefetch block offset $(2 + 1 = 3)$ within the physical page 0xB.

However, before the actual prefetch request is issued, KPC-P must check the status

bit-vectors in the ST to prevent redundant prefetching. If either the corresponding used bit or prefetch bit is already set, KPC-P simply drops the prefetch request. Otherwise, the prefetch is issued and the prefetch bit in the ST is marked to prevent future redundant prefetch requests to that line. Next, KPC-P generates a new, speculative signature based on the demand signature (0x9) and the predicted next cache block delta (+1) (**III** in the figure), creating the first speculative lookahead signature 0x49. KPC-P then continues to recursively index the PT. The lookahead mechanism described here is similar to signature lookahead prefetching [23].

Without a proper throttling mechanism, KPC-P can be too aggressive and pollute the cache through infinite recursion when individual deltas in the PT have 100% confidence. To prevent this from happening, KPC-P calculates a path confidence according to the following formula:

$$C_n = \alpha * C_{n-1} * C_{delta_n} / C_{sig_n} \tag{III.1}$$

where n is the current iteration depth and C_{n-1} was the path confidence of the previous iteration. Here we use the global pefetching accuracy α as a scaling factor, preventing infinite recursion on 100% confident prefetches. Note that α is easily calculated by dividing the number of timely prefetches observed at the ST by the number of prefetch requests at the PT. KPC-P uses two 10-bit counters to track these numbers and calculate α. KPC-P also has a small global history register that records a prefetch request that goes beyond the 4KB physical page boundary. Thus, the global history register is able to provide a signature when there is a new page that is not tracked by the ST. For simplicity, the global history register is not shown in Figure III.6.

The main advantage of KPC-P is that each prefetch request has a proxy of its future use distance in the form of its path confidence C_n. In general, as the depth of lookahead prefetching increases, the confidence decreases. KPC-P exploits this confidence to support

the replacement policy in the LLC. Only when the confidence is higher than the fill level threshold T_F, is a prefetched block also inserted into the L2 (T_F is dynamically adapted by KPC-R as described in Section III.3.2.1). Low confidence prefetches with long predicted use distances stay in the LLC waiting for a prefetch request with higher confidence to pull them into the L2. Second, if a prefetch request is a hit in the LLC, the cache block is promoted within the replacement stack of the LLC only when the prefetch confidence is higher than T_F. Otherwise, a prefetch request with low confidence does not change the replacement state. Low confidence indicates two possible scenarios: a long use distance, or an inaccurate prefetch request. In either case, it is best to avoid cache pollution by not filling a low confidence prefetch into the L2. Furthermore, not promoting on a prefetch hit with low confidence ensures that the LLC can evict these blocks earlier than other blocks with higher priority. Thus, KPC-P uses prefetch confidence to integrate prefetching with the replacement policy, and provides a tool for holistic cache design.

III.3.2 KPC-R: Global Hysteresis Replacement

Here we examine the design and implementation of KPC's global hysteresis replacement algorithm, KPC-R.

III.3.2.1 KPC-R Overview

KPC-R is a low overhead replacement policy that uses a global hysteresis to predict dead blocks by tracking global reuse behavior. Similar to prior work [17, 18], KPC-R associates the cache recency stack with 2-bit re-reference prediction value (RRPV) counters [16] that represent eviction priority. A cache block with a small RRPV counter has low eviction priority whereas a cache block with maximum RRPV may become the next victim in its set. To train the global hysteresis, KPC-R randomly samples 64 sets in the LLC, and duplicates their tags in a separate structure called the LLC sampler. The sampler is managed using true LRU replacement (unlike the real cache, which uses RRPV counters).

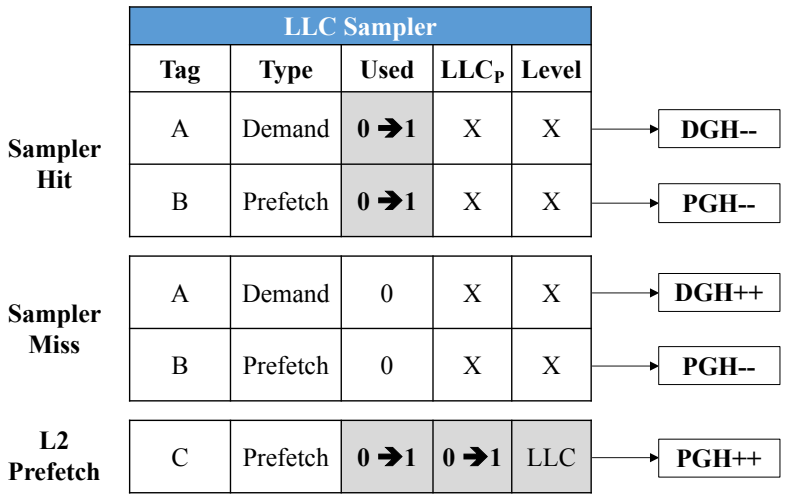


Figure III.8: KPC-R global hysteresis update mechanism

When an LLC access is a hit in the sampler, the global hysteresis decreases, indicating that references during this program phase are more likely to be used again. Because the reuse behavior can be different between demand requests and prefetches, KPC-R has two global hysteresis counters, one for each request type. If there is a sampler miss, KPC-R searches for a victim block using the LRU replacement policy within the sampler. If the victim was never used in the sampler, KPC-R increments the global hysteresis based on its allocation type. Note that this global hysteresis is a per-core counter and not shared by different cores on the same chip.

III.3.2.2 KPC-R Training

Figure III.8 shows the update algorithm of KPC-R. In the figure there are three different training scenarios: Sampler Hit, Sampler Miss and L2 prefetch. The first two cases are straightforward. If there is a hit in the sampler, KPC-R marks the used bit for that cache block and decrements the Demand Global Hysteresis (DGH) or Prefetch Global Hysteresis (PGH) based on its allocation type, indicating greater likelihood that references during this

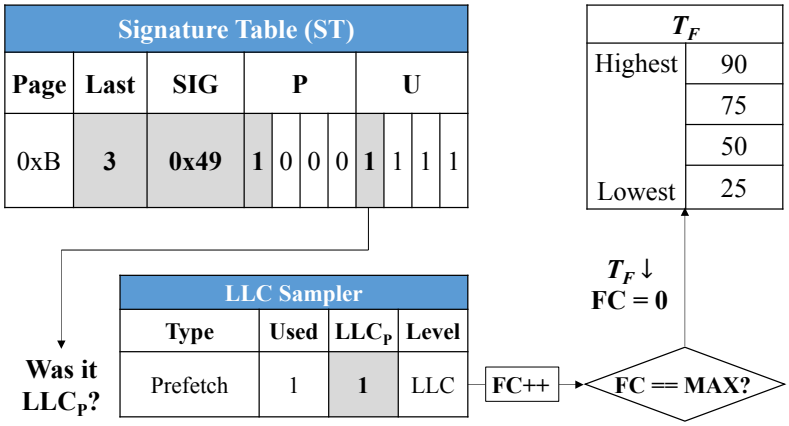


Figure III.9: KPC-R updates fill level threshold for KPC-P

phase will be reused. Once the used bit is set, additional hits to the same cache block do not decrease the DGH.

If there is a miss in the sampler, KPC-R replaces the LRU victim in the sampler. As shown in the figure, if the used bit is set, the victim does not update either the DGH or PGH. Alternately, if the victim is not accessed by either demand or prefetch request, KPC-R increases the global hysteresis value, indicating greater likelihood of references being dead during this program phase.

When there is an L2 prefetch hit in the sampler, KPC-R checks whether this block was allocated by an LLC prefetch through the 1-bit "Level" status in the sampler. If the hit block was filled with an LLC prefetch, KPC-R marks the used bit and LLC prefetch bit in the sampler. Remembering an LLC prefetched block in the sampler allows KPC-R to dynamically update the fill level threshold T_F. Figure III.9 shows how KPC-R updates T_F based on timely prefetch feedback from KPC-P. Because KPC-P has prefetch and used bits in the ST, it can detect timely prefetched cache blocks. When a timely prefetch is detected by the ST, KPC-P probes the LLC sampler to see if this block was initially prefetched

	Demand	Prefetch
Promotion	Always Promote	if ($C_n > T_F$) $\Rightarrow$ Promote
Insertion	if (DGH == MAX) $\Rightarrow$ Dead	if (PGH == MAX) $\Rightarrow$ Dead
Fill Level	Always L2 and LLC	if ($C_n < T_F$) $\Rightarrow$ LLC

Table III.1: KPC-R prediction table

into the LLC and brought up to the L2 later by an additional prefetch request. If so, the additional L2 prefetch request would have been unnecessary if the prefetch fill level was initially set to the L2. Whenever this event is detected by sampler, KPC-R increments the fill level counter (FC) by one. If the FC becomes saturated, then the fill level threshold T_F decreases to allow more prefetches to be filled into the L2 cache.

To avoid T_F becoming too low and KPC-P polluting the L2 cache, we track the global prefetching accuracy α to increase the fill level threshold. A low α value implies that KPC-P is likely to pollute the L2 cache with aggressive prefetching. Therefore, KPC-P increases T_F by one level when α becomes lower than T_F. Thus, KPC-R helps the data prefetcher by setting a dynamic prefetching level and provides another tool for holistic cache design. Table III.1 summarizes the prediction mechanism of KPC-R.

III.3.2.3 KPC-R Placement/Replacement

Based on the insertion and promotion policy described in Table III.1, KPC-R predicts an incoming demand (or prefetch) block to be dead when the DGH (or PGH) is saturated. Dead blocks are inserted to the "LRU" position (RRPV = 3) and do not change other cache blocks' RRPV status. If the global hysteresis is not saturated, the incoming block is inserted to the "near LRU" position (RRPV = 2). Demand requests are always inserted into both L2 and LLC, while a prefetch's fill level is determined by its confidence value.

If there is a cache hit, the promotion scheme is based on its access type and confidence value. For demand requests, KPC-R always promotes reused block to the "MRU" position (RRPV = 0). For prefetch requests, KPC-R promotes a reused block only when its prediction confidence is higher than the current fill level threshold ($C_n > T_F$), as described in Section III.3.1.3. In doing so, KPC-R does not allow extra life time to the cache block reused by low prefetch confidence.

III.4 Evaluation

In this section, we evaluate the KPC system. First, we present the evaluation methodology and compare the performance of KPC with recently proposed replacement algorithms and data prefetchers. We also show in-depth analysis on prefetching coverage with a sensitivity study.

III.4.1 Methodology

We compare KPC with various prefetching and replacement algorithms using the ChampSim simulator, an extended version of the simulation infrastructure used in the 2nd Data Prefetching Championship (DPC-2) [35]. The detailed simulation parameters can be found in Table II.1. We collect SimPoint [36] traces from 16 memory intensive SPEC CPU2006 [37] applications, 3 server workloads from CloudSuite [38], and one machine learning workload trace from mlpack [46] that does collaborative filtering on real world data sets [47]. Since our SimPoint methodology does not work with the server workloads (CloudSuite and mlpack), we instead collect the server workload traces after fast-forwarding at least 30B instructions to get past the benchmark's initialization phase. For all experiments, each trace is warmed up with 200M instructions and simulation results are collected over the next 1B instructions. The baseline replacement policy is LRU replacement for all caches unless otherwise stated. We compare against two prefetch-aware replacement policies, UMO [29] and PACMan [28]. We also compare against two prefetch non-aware replacement policies, EAF [18] and SHiP [17]. Since the original SHiP algorithm does not work well with a data prefetcher as it requires a PC for every insertion, we implement a modified version of SHiP (SHiP+) that uses a single PC value to represent all prefetch requests.

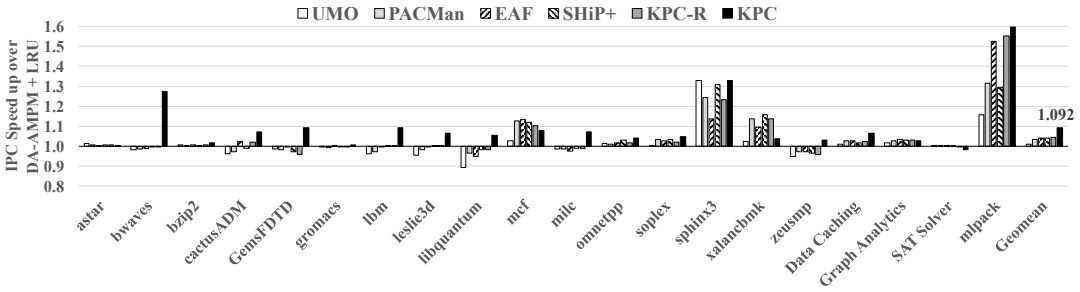


Figure III.10: Single core performance compared to DA-AMPM + LRU

III.4.2 Performance

Single-core Performance: Figure III.10 shows the single-core IPC speedup. All results are normalized to the baseline configuration, where the DA-AMPM prefetcher [29] is used with LRU replacement in the caches. In general, KPC has similar or better performance versus the other cache management schemes. The geometric mean improvement of KPC is 9.2%, 5% higher than the best prior work scheme, the optimized PC-based replacement policy (SHiP+) [17], 5.8% higher than the prior work on prefetch aware mechanism (PACMan) [28], and 8.1% higher than previously proposed unified memory architecture (UMO) [29]. We also plot the performance of our replacement scheme KPC-R combined with the prior work DA-AMPM. Since KPC-R is not co-designed for operation with DA-AMPM, we see that the performance of this combination is only marginally better than SHiP+.

Non-PC-based algorithms such as EAF and KPC-R show less performance improvement for mcf, sphinx3, and xalancbmk. These benchmarks exhibit a large number of LLC misses; the reuse behavior of missing cache blocks in these apps is correlated to a large set of various PCs. In this case, each PC value serves as a unique identifier for reuse prediction and shows higher performance gain than non-PC-based schemes. Still, EAF

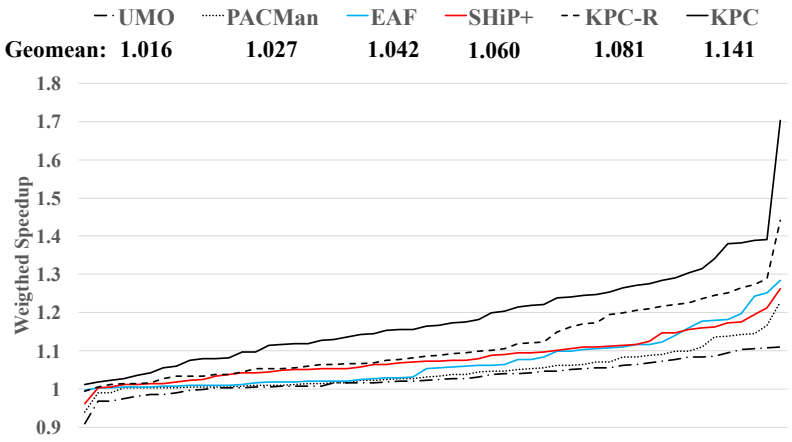


Figure III.11: 4-core multiprogrammed workloads performance

and KPC-R show reasonable performance on these benchmarks. When both prefetching and replacement policy are integrated into one holistic system (KPC), mcf and xalancbmk experience more performance degradation. In this case, both benchmarks benefit from DA-AMPM's aggressive prefetching, which brings more cache lines, achieving more coverage with lower utilization. When confidence is low, KPC dynamically throttles down its lookahead prefetching and has fewer timely prefetches leading to lower performance gains. Aggressive prefetching does not hurt overall performance in a single core environment since only one core is exclusively using the LLC. However, when there are multiple applications competing for the shared resource [40], prior cache management schemes fail to achieve good performance. The multi-core performance of KPC in the next section clearly shows the benefit of dynamic throttling.

Multi-core Performance: Since other schemes have no mechanism to regulate the aggressiveness of the prefetcher based on use and replacement, KPC shows superior performance improvement in a multi-core environment. For multi-core experiments, we randomly generate 54 mixes of 4-core workloads and assign each workload to a different

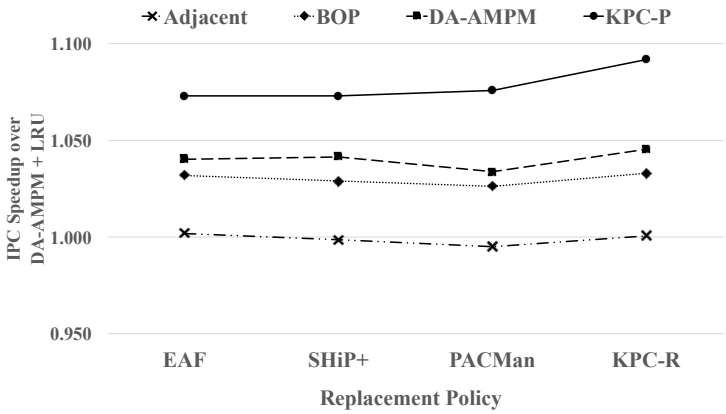


Figure III.12: Various combinations of prefetching and replacement algorithms

core. Figure III.11 shows the normalized weighted speedup of the six different techniques we tested in this work. The graph is sorted by weighted speedup order. On average, KPC achieves a 14.1% speedup and outperforms SHiP+ by 8.1%. Out of the 54 mixes, KPC shows best performance on 52 mixes. For the remaining mixes, SHiP+ or EAF beats KPC by less than 2%. The substantial performance gap between KPC and other techniques is mainly due to the cooperation between KPC-P and KPC-R described in Sections III.3.1 and III.3.2.

KPC-R with other data prefetchers: To further validate the synergy between KPC-R and KPC-P, we also evaluate different types of data prefetchers with various replacement policies. Figure III.12 plots the performance of various combinations between data prefetchers and replacement policies. Here, along with KPC-P and DA-AMPM, we also evaluate an adjacent cache line prefetcher (Adjacent) [48] and Best-Offset Prefetcher (BOP) [24] (BOP was the winner of the 2nd Data Prefetching Competition (DPC-2) [35]). Both prefetchers are offset-based spatial prefetchers which exhibit different characteristic compared to streaming based prefetchers (*e.g.,* DA-AMPM and KPC-P). The performance is normalized to DA-AMPM with LRU. Since UMO shows the least improvement, we have

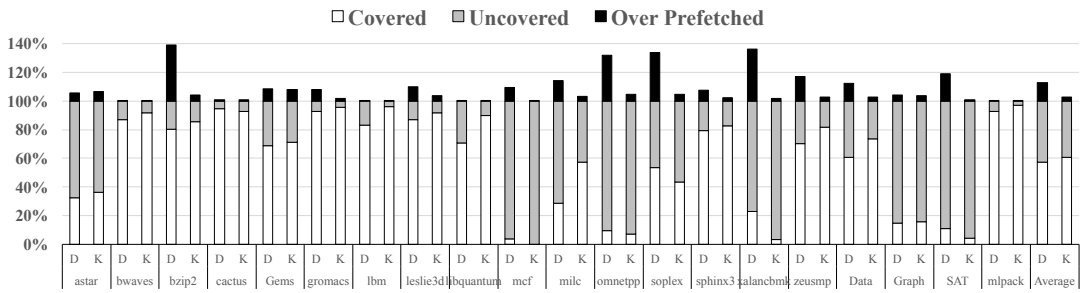


Figure III.13: Prefetching coverage: DA-AMPM vs. KPC-P

removed it from this figure. Figure III.12 shows that KPC-R still achieves great performance versus other techniques regardless of prefetching algorithms. In particular, KPC-R achieves the best performance when it is combined with KPC-P. KPC-R adds 3.8% performance on the top of KPC-P while PACMan shows the second best performance of 2.3% above KPC-P with LRU. To summarize, the global hysteresis replacement scheme works well for both spatial and streaming based prefetchers, however, its gain is maximized when KPC-R is used with KPC-P due to the synergistic effect of holistic cache management.

III.4.3 Analysis

Prefetching Coverage: Figure III.13 shows prefetching coverage of DA-AMPM and KPC-P. Each prefetcher is labeled by its first letter. On average, KPC-P covers 61% of on-chip cache misses with 3% over prefetched blocks while DA-AMPM covers 59% misses with 13% over prefetches. Over prefetches represent the sum of prefetches never used prior to eviction, caused by aggressive prefetching. Compared to KPC-P, DA-AMPM produces 10% more over prefetched blocks. In particular, we can see that DA-AMPM generates a higher fraction of over prefetching for mcf, xalancbmk, and SAT solver, where DA-AMPM achieves greater performance. Note that for these benchmarks, the actual number of prefetches is also higher for DA-AMPM. While this does provide some gain

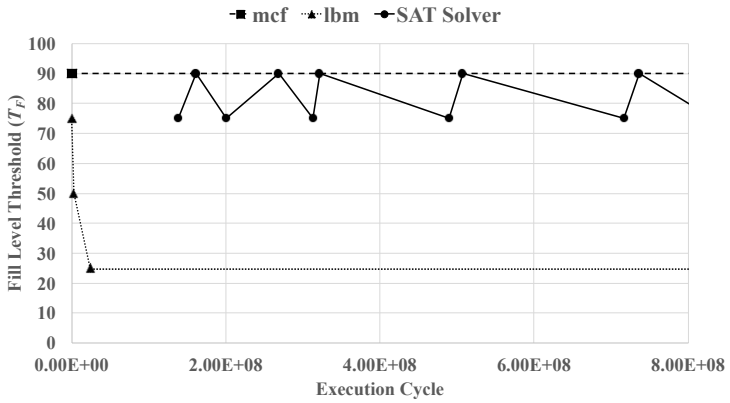


Figure III.14: Dynamic adaptation of fill level thrshold T_F

for these benchmarks, the over prefetching becomes a serious issue when multiple applications contend for shared resources such as LLC capacity and memory bandwidth, as shown in Figure III.11. Since KPC is designed to adapt prefetch placement and aggressiveness by closely integrating the prefetcher and replacement policy, KPC exhibits less performance gain for single-core but achieves much higher benefit in a multi-core environment.

Dynamic Fill Level Thrshold: Figure III.14 shows the dynamic adaptation of the fill level threshold T_F over program execution. As we see, each benchmark prefers a different T_F value. For example, KPC lowers T_F for lbm since most of prefetches are useful in the L2 cache. The timely prefetch feedback from KPC-P to KPC-R quickly adjusts the T_F value. On the other hand, KPC does not change T_F for mcf since its reference pattern is unpredictable. For SAT Solver, T_F frequently changes due to program phase, based on its global prefetching accuracy. Thus, KPC dynamically adapts its fill level threshold to each individual application.

Global Hysteresis Sensitivity: We also investigated the performance sensitivity of the global hysteresis to its counter bit-width (figure removed for brevity). We changed the width of global hysteresis from 1-bit to 10-bits and measured the geomean performance.

Structure	Entry	Component	Storage
Signature Table	256	Valid (1 bit) Tag (16 bit) Last offset (6 bit) Signature (12 bit) Prefetch (64 bit) Used (64 bit) LRU (8 bit)	43776 bits
Pattern Table	512	C_{sig} (4 bit) C_{delta} (4*4 bit) Delta (4*7 bit)	24576 bits
LLC Sampler	64	Valid (1 bit) Tag (16 bit) Type (1 bit) Used (1 bit) LLC_P (1 bit) Level (1 bit) LRU (4 bit)	1600 bits
Hysteresis	1 1	DGH (3 bit) PGH (3 bit)	3 bits 3 bits
Misc.	-	Registers	284 bits
43776 + 24576 + 1600 + 6 + 284 = 70242 bits ≈ 8.57 KB			

Table III.2: KPC storage overhead

Surprisingly, we find that, for single-core, the performance of KPC is very insensitive to the width of the global hysteresis counter. Even for a 1-bit counter, performance drops by only 1.2%. For multi-core, however, we found that using a counter width below 2-bits suffers from substantial performance degradation. Though the global hysteresis is a per-core counter, multiple applications put additional memory pressure on the LLC and frequently toggle the prediction of a 1-bit global hysteresis, causing performance degradation. The final design of KPC uses 3-bit global hysteresis because there is no visible difference in performance above 3 bits.

Storage Overhead: Table III.2 shows the storage overhead of KPC. All together, KPC requires a modest 8.57KB of state per core, with only 2% of the overhead coming from KPC-R. Most of the storage overhead lies with KPC-P, since KPC-R only uses a small sampler and two narrow hysteresis counters per core. Table III.3 presents a storage com-

Prefetcher	Replacement	Total Storage
DA-AMPM: 4.8 KB	UMO: 20 bits	≈ 4.81 KB
	PACMan: 30 bits	≈ 4.81 KB
	EAF: 32 KB	≈ 36.8 KB
	SHiP: 7.18 KB	≈ 11.98 KB
	KPC-R: 0.23 KB	≈ 5.03 KB
KPC-P: 8.34 KB	KPC-R: 0.23 KB	≈ 8.57 KB

Table III.3: Storage overhead comparison

parison between KPC and previous cache management schemes. Note that the storage overhead of KPC is comparable to existing state-of-the-art replacement algorithms while providing higher performance.

Moreover, unlike PC-based schemes, KPC does not require extra logic/wires to propagate PC values through from the instruction fetch unit, to the load store queue, and then the entire cache hierarchy. Both UMO and PACMan rely on set dueling [15] that only requires two or three global counters. In doing so, their storage overheads are very small but come with the cost of low performance improvement. In particular, in multi-core environments, set dueling based policies show less performance gain than others because the learning overhead from set dueling monitors grows as the number of cores increases [28].

III.5 Summary

Kill-the-PC reconstructs program behavior in the cache hierarchy by taking a holistic approach to cache management. Tightly integrating L2 prefetching and LLC management into one unified solution, KPC is effective at making sure data is as close to the processor core as possible when it is needed by a demand access. In addition to being an effective prefetcher for complex delta patterns, the prefetching component, KPC-P, feeds confidence information about each individual prefetch to the LLC replacement component, KPC-R. A low confidence prefetch is less likely to interfere with the contents of the LLC, and as confidence in that prefetch increases, its position within the LLC replacement stack is solidified, and it eventually is brought into the L2 cache, close to where it will be used in the processor core. In addition to effectively predicting dead LLC blocks by observing program phase behaviors, KPC-R also gives feedback to KPC-P to help decide on the optimal fill level for prefetches.

While KPC-P and KPC-R can each stand on its own as an effective solution within its own domain, the combination of both is more effective than replacing either component with a state-of-the-art solution. KPC provides a 9.2% performance benefit versus a baseline system with a top-of-class prefetcher, besting the nearest competitor by 5%. Further, across our suite of multicore mixes, KPC improves performance by 14.1% versus a prefetching+LRU baseline and extends its lead over the best of class competitor to 8.1%.

CHAPTER IV

DYNAMIC MEMORY REALLOCATION IN VIRTUALIZED SYSTEM*

In this chapter, we propose a novel dynamic memory management policy called Memory Pressure Aware (MPA) ballooning. MPA ballooning dynamically allocates memory resources to each virtual machine (VM) based on the current memory pressure regime. Moreover, MPA ballooning proactively reacts and adapts to sudden changes in memory demand from guest VMs. MPA ballooning requires neither additional hardware support, nor incurs extra minor page faults in its memory pressure estimation. We show that MPA ballooning provides an 13.2% geomean speed-up over the current ballooning techniques with hypervisor caching technology Tmem [49] across a set of application mixes running in guest VMs.

IV.1 Introduction

To dynamically adjust memory allocation, Waldspurger [50] introduced the "ballooning" technique as a kernel module driver with VMware ESX Server [51]. The memory allocation is based on a share-based allocation scheme [52] where the share of VM is determined by its memory utilization. When the host machine is under memory pressure and needs to reclaim memory from guest VMs, reclamation is done by measuring the portion of idle memory in each guest VM. To measure the idle memory, VMware ESX tracks randomly selected pages by invalidating their associated TLB entries. Subsequent accesses to the sampled pages trigger TLB misses and increase the count of touched pages. The fraction of inactively accessed memory is estimated from the ratio of untouched pages to sampled pages. Similarly, recent studies [53, 54, 55] force TLB misses to estimate the

*Reprinted with permission from "Dynamic Memory Pressure Aware Ballooning" by J. Kim, V. Fedorov, P. V. Gratz, and A. L. Narasimha Reddy, 2015. Proceedings of the 2015 International Symposium on Memory Systems, Pages 103-112, Copyright 2015 by ACM

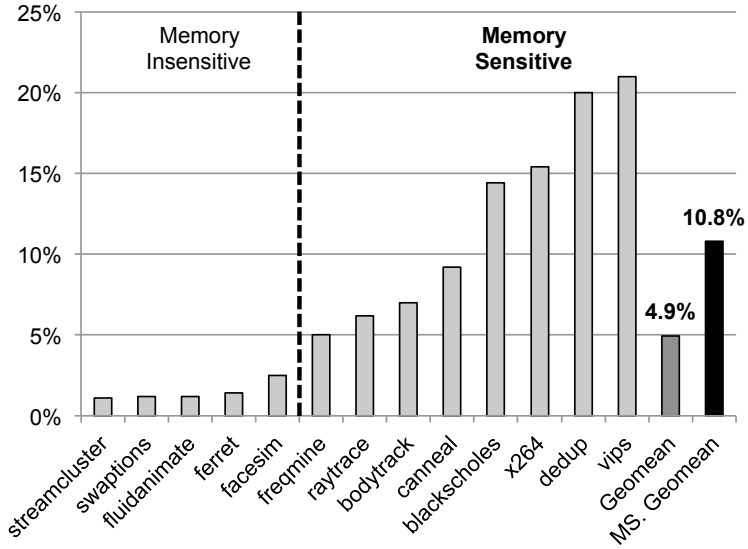


Figure IV.1: Performance degradation of ballooning with Tmem

working set size of each VM. Invalidating TLB entries, however, incurs a substantial performance overhead in virtualized systems, since it forces a context switch and requires multiple memory accesses to refill the entries [56].

Further, the current memory allocation policies bear significant performance overhead due to insufficient knowledge about global memory pressure. Hypervisor exclusive cache [54] or Tmem [49], manage an additional layer between physical RAM and the disk, to cache pages evicted by the guest VMs in a common pool controlled by the hypervisor. However, they aggressively and obliviously claw-back file cache pages from VMs to the hypervisor, even when sufficient memory is available in the host machine. This causes substantial performance impact due to excess page movement and page faults regardless of global memory pressure. Figure IV.1 shows the overhead of ballooning with Tmem on individual applications from the PARSEC suite, running in a guest VM *with no other load in the system*. In the figure, we observe that ballooning with Tmem results in 10.8% of performance overhead compared to a system without ballooning support on memory

sensitive applications. In contrast to Tmem, VMware ESX adopts the concept of global memory pressure and activates ballooning driver only when the host free memory drops towards statically predefined threshold [51]. This approach also degrades the memory utilization and responsiveness of ballooning because most of physical memory is likely to be consumed by the guest VMs without actually being used.

Ideally, the memory allocation strategy for virtualized systems should *adapt dynamically to the global memory pressure with minimal overhead*. This is the goal of MPA ballooning. The key concept of MPA ballooning is that the hypervisor unobtrusively measures the current system memory pressure and adaptively changes memory allocation policy. For example, if the host machine is under low memory pressure, MPA ballooning allows guest VMs to have additional memory cushion so that they can reduce the overhead of acquiring pages from the hypervisor. On the other hand, if the host machine is under heavy memory pressure, the hypervisor perceives increased memory pressure and reclaims the inactive pages from guest VMs. Unlike prior works [55, 51] which considered memory cushion only when the host machine has enough free memory, MPA ballooning dynamically changes the amount of base and bonus memory with respect to the different memory pressure regimes. We implement MPA ballooning in Xen hypervisor and Linux kernel with minor modifications. We show that the MPA ballooning allows guest VMs to share memory across all ranges of memory pressure, thus enabling their wider adoption.

MPA ballooning makes the following major contributions:

- We identify different regimes of memory pressure in a system and employ this identification to drive the memory allocation and sharing policies in a virtualized environment.

- We develop mechanisms to measure the memory pressure state of the virtualized system to guide the memory allocation policy. These mechanisms require neither

additional hardware support, nor incur significant performance overhead in memory pressure estimation.

- We implement MPA ballooning and evaluate it on real workloads. The proposed technique improves performance by 18.2% with multiple VMs repeating single applications. In more realistic scenario with high memory pressure, MPA ballooning improves performance by 13.2% with multiple VMs repeating random ordered applications. All experiments were evaluated under various memory pressure on the host system.

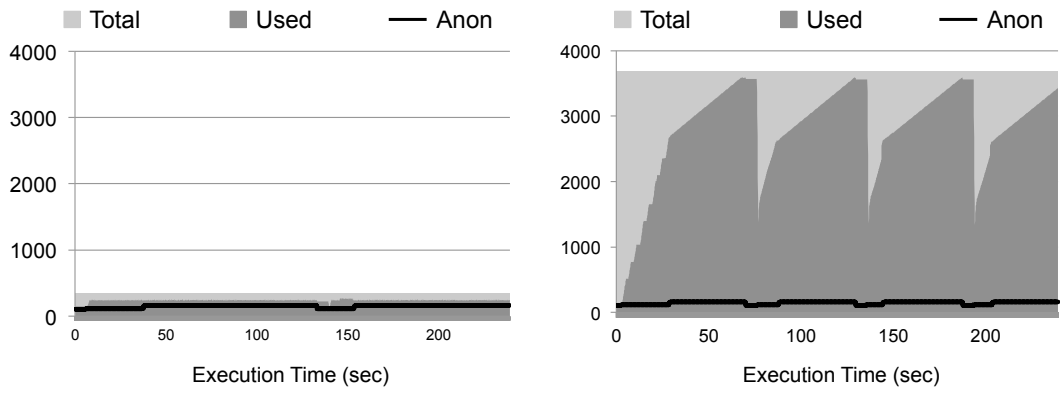
IV.2 Design Motivation

We observe that traditional ballooning drivers incur a high performance penalty because the algorithm is oblivious to system conditions and often incorrectly sets the memory target for the guest VM. Prior work in ballooning are neither capable of reacting fast enough to deal with rapidly changing application memory demands, nor adaptive to the system memory pressure regimes described in Section IV.3.

IV.2.1 Adaptation to System Conditions

Typically the ballooning driver chooses its target based on memory resource requirements. For instance, the ballooning driver implemented in Linux as part of Xen, sets the balloon target equal to the sum of the malloc'ed memory at any given time. On the other hand, memory reclamation policies can be classified into two categories: aggressive and speculative. Hypervisor exclusive cache [54] and Tmem [49] belong to aggressive management policies. These approaches implement a hypervisor-level cache which stores evicted pages from guest VMs to reduce the number of disk I/Os in near future. A critical problem with aggressive reclamation, however, is that the guest VMs are forced to evict file cache pages without considering current system state.

Figure IV.2 illustrates how a typical ballooning driver with Tmem results in performance degradation by evicting and reloading file pages. In the figure, the lightly shaded area $Total$ represents the memory assigned by the hypervisor to the VM, while the darker area $Used$ is the amount of memory actually being used. The application is *vips*, an image processing application from the PARSEC suite [57], which reads a large input image file using memory mapped I/O. After a file has been loaded, all image operations are applied to the loaded input file. In this experiment *vips* is iterated four times to analyze the overhead that comes from accessing evicted file pages in Tmem. Since the input image mapped in memory is identical to the content of physical disk, they are not accounted as



(a) Ballooning + Tmem

(b) Ballooning off

Figure IV.2: False balloon target due to clean pages in *vips* from the PARSEC suite

$Anon$ (shown in the black line). The baseline ballooning driver, therefore, does not change the balloon target (Figure IV.2a) though the actual memory usage is much larger than the target. Consequently, the guest OS evicts file pages to make more space to read the input image file. Evicted pages are copied into the Tmem managed cache area. Each time *vips* reloads the file, the pages in the Tmem cache are accessed again so that the hypervisor can avoid disk I/O, resulting in another copy back (reloading) to the guest VM. Figure IV.2b shows the actual memory usage of *vips* running in a VM without ballooning. In this case, each time *vips* reloads the file, it reuses the file pages maintained in a guest VM's memory from the previous iteration. The file pages kept with ballooning off is shown as $Used$ above the $Anon$ line in Figure IV.2b.

Comparing the execution time of Figure IV.2a and IV.2b, reloading pages stored in the Tmem cache results in 20% performance overhead. It is clear that the hypervisor should not reclaim pages aggressively from VMs. In other words, if actively used pages can be kept in the guest VMs, the performance degradation will significantly decrease. Based on this motivation, MPA ballooning adaptively changes memory allocation and deallocation

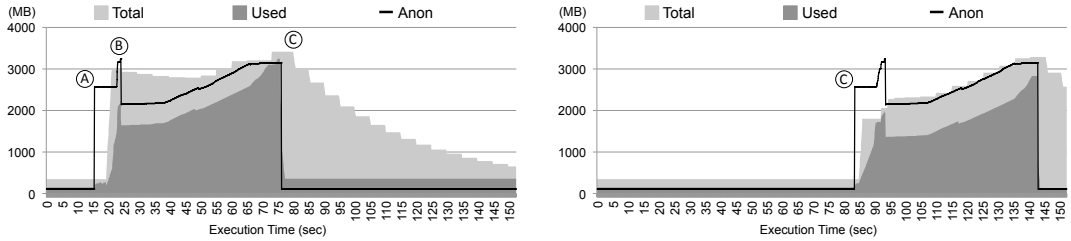
policy according to the memory pressure state.

Other works based on speculation [50, 53, 55, 58] are less aggressive than Tmem because they let the guest VMs keep pages based on memory access speculation without implementing a hypervisor-level shared cache. Only when the memory pressure exceeds a certain threshold, these pages are *slowly* reclaimed by the hypervisor. This approach, however, can suffer performance degradation when the host machine is under memory pressure, particularly when the memory consumption is changing rapidly. Under these conditions page reclamation should be accelerated such that pages can be reallocated to the VM which can use it most effectively.

IV.2.2 Slow Reclamation and Reallocation

To address the performance cost of latency in memory reclamation and reallocation, changes in the memory demands of each guest VMs should be communicated to the hypervisor immediately. With more accurate and timely information about guest VM memory needs, the hypervisor can better manage each VM's memory allocation based on a view of current system demands. Current ballooning drivers [54, 49] do not react dynamically with the varying system conditions since the activation period is statically set by the hypervisor. For example, Xen self-ballooning driver is invoked every five seconds to update the balloon target [59]. Thus, sudden changes of memory demand are not processed until the end of the current five second period, potentially impacting performance. Similarly, the VMware ESX Server uses sampling to activate memory reclamation. By default, VMware ESX Server samples 100 guest pages for each 60-second period [51].

To analyze the impact of slow response in memory allocation and deallocation process, we set two VMs to execute *dedup* at different time. Figure IV.3a shows VM1 starting *dedup* at 15 seconds and Figure IV.3b shows VM2 starting the same application at 85 seconds. In this experiment, the host machine memory is limited to 4.5GB so that we can



(a) VM1 executes dedup at t = 15

(b) VM2 executes dedup at t = 85

Figure IV.3: Slow response to memory allocation and deallocation

clearly observe memory allocation and deallocation process.

Point **A** and **B** in Figure IV.3a illustrate how latency in responding to dynamic memory demand can affect memory allocation. At point **A**, *dedup* requests a large chunk of memory, causing $Anon$ to instantly jump to 2.5GB. This occurs in the middle of the five second interval, thus the ballooning driver does not respond to this change and the $Total$ memory assigned to the VM does not change. Meanwhile, *dedup* rapidly attempts to use the memory it requested leading to major page faults because insufficient memory is assigned to the VM. Point **B** in Figure IV.3a illustrates a further hazard of latency in responding to memory demand. Here, $Anon$ varies dramatically within the five-second sampling period due to a large allocation, use and deallocation. The ballooning driver is not aware of any of these changes and does not adjust the target.

On the contrary, Point **C** illustrates how slow memory deallocation impacts the other VMs running. At 85 seconds, VM2 starts running *dedup* and its $Anon$ jumps up to 2.5GB seeking for more memory. However, VM1 slowly returns memory to the hypervisor which in turn slows down the memory reallocation to VM2. Therefore, VM2 cannot achieve enough memory to execute *dedup* and suffers from major page faults. Both Xen hypervisor and VMware ESX Server adopt slow deallocation process inspired from control theory and networking domain [50]. To control the rate of variation in reclaim process, they

both adopt hysteresis value. In other words, the memory target is determined not just on current data but also on past data stored in the system. However, this approach only covers local memory usage in each VM and does not consider global memory pressure state. As a result, the memory reallocation speed does not adapt to other VM's memory demand or global memory pressure status.

Naïvely, one might consider increasing the sampling frequency of the ballooning driver, however, this approach could incur high overheads from the additional hypercalls. Instead, we propose to capture the exact moment when the memory demand increases or decreases unexpectedly and invoke ballooning driver without delay.

IV.2.3 Working Set Estimation Overhead

Several prior work memory allocation policies [50, 53, 55, 54] have adopted a memory trap technique to sample the memory accesses by setting a privilege bit in a page table entry. In doing so, the hypervisor samples memory accesses to build a page miss ratio curve which can provide an estimation of working set size (WSS). These techniques, however, often come at a significant performance cost due to the required TLB and page walk cache entry invalidations. For example, the TLB invalidation instruction in x86 architecture invalidates all entries in all page walk caches associated with the current PCID, regardless of the virtual addresses to which they correspond [60]. In virtualized environments, invalidating TLB and page walk cache results in more memory accesses than that of native machine since each pointer in the guest page table must be translated through the hypervisor page table [56].

Instead of estimating the memory footprint using high-overhead TLB and page walk cache invalidations, in MPA ballooning, we propose to exploit information directly exported by the guest OS's to the hypervisor. Specifically, as part of the ballooning driver, the guest OS reports the active and inactive list up to the hypervisor [61]. Although this

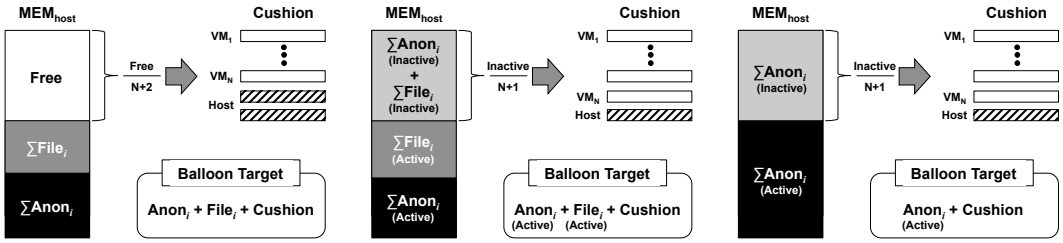
approach does come at the cost of a limited number of additional hypervisor calls, these calls have much less performance overhead than the invalidations used in the ampling approaches. We note that this use of back-channel information is similar to the information passing utilized by the Xen hypervisor's guest OS ballooning drivers, however, as we discuss in the next section, we greatly extend the interface to provide a much more detailed and timely picture of the guest OS's memory usage state.

IV.3 Design

The proposed MPA ballooning addresses the oblivity to memory pressure shown in existing ballooning techniques, as discussed in the previous section. The overall design of MPA ballooning can be divided into two components. First, MPA ballooning implements an adaptive memory cushion which dynamically changes its size based on the system memory pressure. Second, MPA ballooning implements an instant response mechanism which dynamically reacts to changes in memory pressure through accelerated memory reclamation and reallocation. Similar to Ginkgo [58], the approach devised in MPA ballooning is hypervisor-agnostic in that the framework only requires a ballooning driver to dynamically control the memory size of a VM, which is already available in most of hypervisors such as VMware ESX Server [51], Xen [31], and KVM [62].

IV.3.1 Adaptive Memory Cushion

As noted in Section IV.2, the hypervisor must dynamically adapt to the current memory pressure in order to avoid unnecessary page movement and disk I/O. We design an adaptive memory cushion in MPA ballooning so that the memory can be efficiently reallocated among guest VMs. The adaptive cushion is managed by the hypervisor and the size of cushion dynamically varies as the memory pressure changes. Figure IV.4 describes how the MP aware ballooning calculates the memory cushion and balloon target based on different memory pressure regimes. In the *Low pressure* regime (Figure IV.4a), the host machine does not suffer from memory pressure because the total memory demand is always lower than MEM_{host}. Therefore, the hypervisor splits $Free$ memory with $N + 2$ instances where N is the number of guest VMs. Each guest VM then receives a cushion slice of $Free/(N + 2)$ above their local memory demand ($Anon_i + File_i$). This cushion allows the guest VMs to absorb dynamic changes in memory demand in the VM without further intervention from the hypervisor. The balloon target for each guest VM is set by

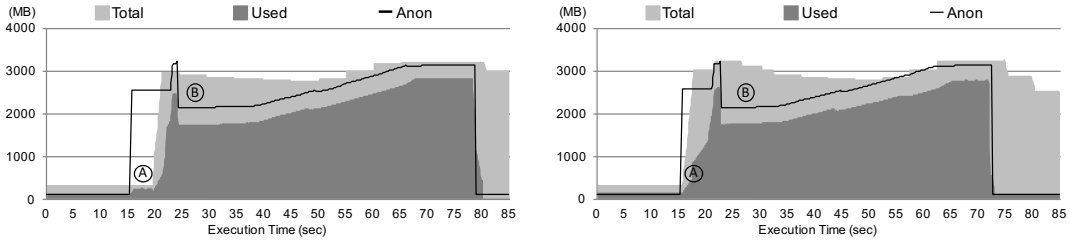


(a) Low pressure (b) Mild pressure (c) Heavy pressure

Figure IV.4: Adaptive memory cushion.

adding the adaptive memory cushion on the top of local memory demand. We reserve two memory cushions of $Free/(N+2)$ for the hypervisor as spare memory that can be quickly allocated to a given guest OS in the event of a sudden demand.

When the total memory demand exceeds the MEM_{host} limit, the host machine enters the *Mild pressure* regime and the memory cushion must be rebalanced. Since active pages are more likely to be used by guest VMs, while inactive pages are less likely, MPA ballooning uses the inactive memory in the guest VMs as an adaptive cushion in the *Mild pressure* regime. As shown in Figure IV.4b, MPA ballooning divides up inactive pages with $N + 1$ instances of memory cushion in *Mild pressure* regime. Rebalancing with $N + 1$ cushions forces guest VMs to yield a portion of its inactive pages to other VMs. These reclaimed pages can be used for guest VMs with high active memory demands. By reclaiming inactive pages from guest VMs, the system may return to the *Low pressure* regime. If the guest VMs do not change their local memory demands, the retrieved memory remains *Free* in the hypervisor. The VMWare ESX Server also falls back to the low memory pressure state when reclaiming pages from guest VMs when the host free memory is below 6% of its total memory. In this scheme, however, the memory pressure state is purely based on host memory size [51], thus this policy in the VMWare ESX Server can lead to underutilized memory. On the other hand, MPA ballooning reclaims and rebalances pages based directly



(a) Slow response

(b) Improved response

Figure IV.5: Improved response time with trigger

on guest VM's memory demands.

If the total memory demand is overwhelmed by $\Sigma Anon_i$, file pages are mostly evicted and the host machine may see performance degradation due to increased file I/O. In this regime, an adaptive memory cushion is built by dividing up the inactive anonymous pages. Note that the size of cushion dynamically adapts to varying system conditions because the cushion determined by *inactive* portion of memory.

IV.3.2 Memory Reallocation Trigger

Compared to the guest VM scheduling time slice, which is usually 30ms in Xen hypervisor [63], the five second response interval of ballooning driver is relatively slow. VMware ESX Server has better response since it activates ballooning driver when the hypervisor detects 6% threshold of host free memory. However, the detection mechanism, based on a 60 second sampling period [51] potentially slows down the response of memory reallocation process. MPA ballooning resolves this problem by implementing a memory use threshold trigger which immediately activates the ballooning driver rather than waiting for the next response interval. Thus, memory reallocation is accelerated and the guest VM can avoid major page faults.

Figure IV.5 shows the effect of our memory reallocation trigger. Since an abrupt change in $Anon$ occurs within the response interval (five seconds here), the ballooning

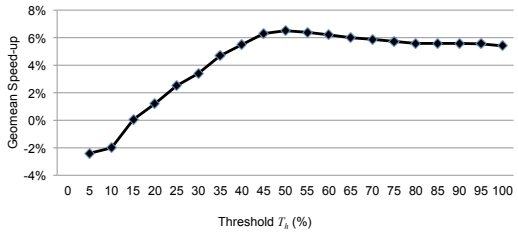


Figure IV.6: Performance analysis on trigger threshold

driver cannot detect the change in malloc'ed memory and fails to set a correct target in Figure IV.5a. As a result, the amount of memory given to a guest VM ($Total$) remains unchanged at Point **A**, starving the guest VM of memory to meet the sudden demand and leading to performance loss due to major page faults. In MPA ballooning, shown in Figure IV.5b, the memory reallocation trigger detects the change in memory demand and immediately activates ballooning driver. Thus MPA ballooning, improves reaction time dramatically and reduces the gap between $Total$ and $Anon$. The reallocation trigger activates ballooning driver based on a threshold formula as follows.

$$|Total_c - Anon_c| \times T_h < R_{size} \tag{IV.1}$$

Here, $Total_c$ is the current total memory given to a particular guest VM, $Anon_c$ is the current malloc'ed memory prior to the coming change and R_{size} is size of change in $Anon_c$ requested. If R_{size} is larger than $T_h\%$ of the difference between $Total_c$ and $Anon_c$, the ballooning driver is immediately triggered via hypervisor call. As shown in Figure IV.6, we examined a wide range of threshold values and $T_h = 50\%$ was empirically determined to be optimal.

IV.3.3 Adaptive Hysteresis

Typically, increasing memory allocation in MPA ballooning occurs quickly via the re-allocation trigger, while decreasing memory allocation is relatively slow so that the guest OS can adapt to any unexpected change of its local memory demand. When the host machine is under memory pressure (*i.e., Mild* or *Heavy pressure*), however, the hypervisor should control the hysteresis value so guest VMs with inactive pages can quickly return pages to other VMs with high memory demands. As noted in Section IV.2.2, if hysteresis value does not adapt to global memory pressure, the slow memory reallocation will degrade overall system performance. MPA ballooning uses the following formula to dynamically adapt its hysteresis to the current memory pressure regime:

$$H_{MP} = H_{default} \times \frac{\text{Global Inactive Page}(GIP)}{Inactive_i} \tag{IV.2}$$

$$GIP = \frac{\Sigma Inactive_i}{\text{Number of VMs}} \tag{IV.3}$$

Equations (2) and (3) are inspired by prior work [50] which considered the amount of idle pages for page reallocation. Similarly, MPA ballooning dynamically changes the hysteresis value for each guest VM by comparing its inactive page count to the global average. If the hysteresis value H_{MP} is high, the system responds slowly to both increasing and decreasing memory requests. Alternately, when the hysteresis value is low, the algorithm responds quickly to requests in both directions. In the *Low pressure* regime, H_{MP} is set to a high default value ($H_{default}$) because the host machine has enough memory to cover requests without rapid hypervisor intervention. When the memory pressure regime enters the *Mild or Heavy pressure* regime, MPA ballooning compares GIP which represents the

average number of inactive pages across the all VMs with the guest VM's inactive page count ($Inactive_i$). If $Inactive_i$ is higher than GIP, it implies that the reclamation process can be accelerated in VM_2 because it has more inactive pages than average. On the other hand, if local $Inactive_i$ is less than the global average, we maintain the previous H_{MP} value. By comparing the local inactive pages with the global average, the MPA ballooning accelerates memory allocation and deallocation process. Note that H_{MP} is related to the speed of memory management and has nothing to do with the size of allocation or deallocation.

IV.3.4 Implementation

The MPA ballooning is a pure software approach and requires minimal changes to the ballooning driver and Hypervisor to implement. To evaluate the technique we implemented MPA ballooning in the Linux kernel and Xen hypervisor. Table IV.1 shows the number of modified/additional lines of code required to implement each component of MPA ballooning. Since the memory reallocation trigger is activated by the guest OS side, it does not require any change in the hypervisor. To set a new balloon target with an adaptive memory cushion, the MPA ballooning requires only one additional hypercall from the guest OS. Furthermore, the design is hypervisor-agnostic because the implementation only requires a ballooning driver which is already widely adopted in most hypervisors.

Component	Linux Kernel	Hypervisor
Adaptive Cushion	15 lines	120 lines
Reallocation Trigger	25 lines	-
Adaptive Hysteresis	20 lines	20 lines
Total	60 lines	140 lines

Table IV.1: Design overhead of MP aware ballooning

Domain	Component	Configuration
Host	CPU	Intel Xeon E5-2420 1.90GHz
	# Cores	6
	L1I & D cache	32KB 8-way
	L2 cache	256KB 8-way
	Shared LLC	15MB 20-way
	DRAM	32GB
	VM Technology	VT-x, VT-d, EPT
	Host OS	Ubuntu Linux 12.04.3
	Hypervisor	Xen 4.2.3
Guest	Virtualized CPUs	4
	DRAM/guest	4GB
	Guest OS	Ubuntu Linux 12.04.3

Table IV.2: Baseline configuration

IV.4 Evaluation

In this section we first describe our experimental methodology. We then explore the MPA ballooning's performance under different workloads.

IV.4.1 Methodology

All experiments were performed in real hardware, on an Intel Xeon E5-2420 1.90GHz machine. The baseline hardware configuration is shown in Table IV.2. All experimental results are normalized to an ideal, non-memory constrained case where the hypervisor can fully accommodate the sum of each VM's configured main memory (4GB each). Additionally, the ballooning driver is disabled so that the hypervisor does not reclaim any pages from the guest VMs. In this configuration, a minimal number of page faults and file I/O requests to load pages from disk to memory occur during the very first execution. By comparing the speedup against the non-memory constrained, ideal case, we clearly show that Tmem causes significant performance degradation and how MPA outperforms Tmem by reducing major page faults and file I/O.

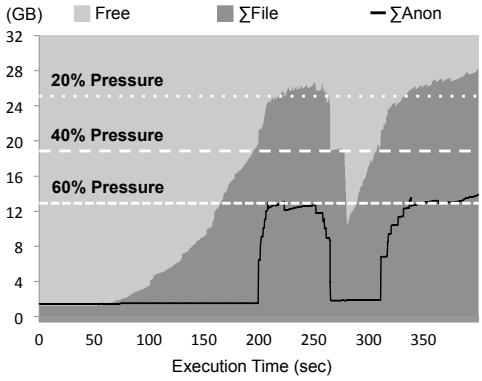


Figure IV.7: Create memory pressure by restricting MEM_{host}

MPA ballooning is evaluated under three different memory restriction models: 20%, 40%, and 60% pressure (20% represents 20% less DRAM is available than the sum of configured guest VM's main memory). These models are illustrated in Figure IV.7, each dashed line represents memory restriction we applied to the physical memory. In each restriction model, the hypervisor experiences different memory pressure regimes described in Figure IV.7. For example, with 20% restriction model, the hypervisor always stays in *Low* pressure while the 40% restriction model experiences transition from *Low* to *Mild* regime. Similarly the 60% model experiences all three memory pressure regimes. With greater restriction amounts, there is a greater likelihood to be in the *Mild* or *Heavy* regimes.

MPA ballooning is evaluated using applications from the PARSEC suite [57] of shared memory multiprocessor benchmarks executing in guest VMs. The PARSEC suite includes benchmarks selected from different computation fields and the memory usage of each benchmark is diverse enough to represent applications running in large scale servers. We test MPA ballooning in two different test scenarios. In the first experiment, we execute 8 VMs simultaneously, with each VM repeatedly executing a single benchmark, selected as one of the memory sensitive applications shown in Figure IV.1. The second experiment runs multiple VMs where each VM repeats multiple applications mixed in random order.

MIX	Benchmark 1	Benchmark 2	Benchmark 3
VM1	bodytrack	dedup	x264
VM2	blackscholes	raytrace	canneal
VM3	freqmine	dedup	x264
VM4	bodytrack	blackscholes	x264
VM5	vips	raytrace	canneal
VM6	bodytrack	vips	canneal
VM7	freqmine	vips	dedup
VM8	freqmine	blackscholes	raytrace

Table IV.3: Randomized mixes

To avoid biased selection on a certain type of benchmarks, we generate 8 randomized mixes of the PARSEC suite (Table IV.3). Between each iteration, the VM executes a *sleep* time randomly selected from 0 to 20 seconds in order to evaluate the response time of Tmem and MPA ballooning.

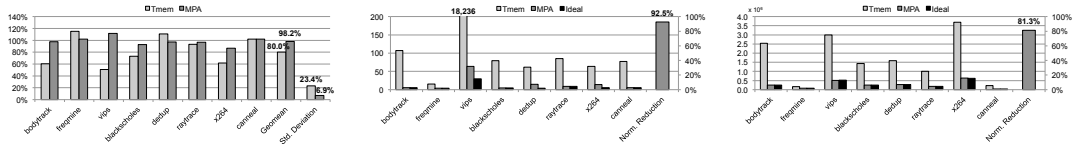
IV.4.2 Repeating Single Application

We first evaluate a scenario where each guest VM iterates a specific application. Memory sensitive applications are assigned to guest VMs and the experiment is stopped when all applications have executed more than five instances. The average execution time of each of those first five instances is used for performance evaluation.

As Figure IV.8a shows, MPA ballooning shows performance nearly identical to that of the non-memory constrained system (Ideal). Compared to Ideal, MPA shows less than 2% slowdown, 18.2% better than that of Tmem across all mixes under the 20% physical memory restriction model. With this relatively small memory restriction, the host machine remains in *Low pressure* regime. As a result, the adaptive cushion of MPA ballooning is able to absorb both anonymous and clean page demands, thereby dramatically reducing the amount of page movement between the guest VM and Tmem (seen as major page faults in the guest). Here, the reduction in major page faults (Figure IV.8b) and file read operations

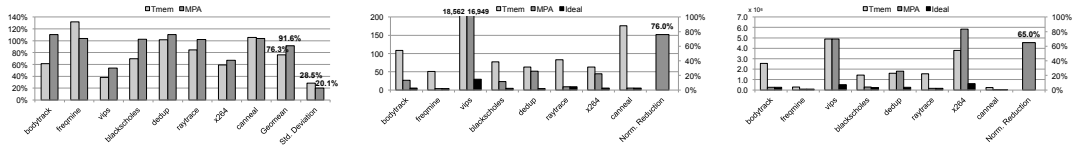
(Figure IV.8c) corresponds to the speedup of MPA ballooning. The right most bar of Figures IV.8b and IV.8c, *Norm. Reduction*, shows the normalized geomean reduction in faults and file reads respectively, across all benchmarks.

There are exceptions where reduced file I/O does not match the speedup value. For example, with Tmem, *freqmine* and *dedup* generate more page faults and file reads but



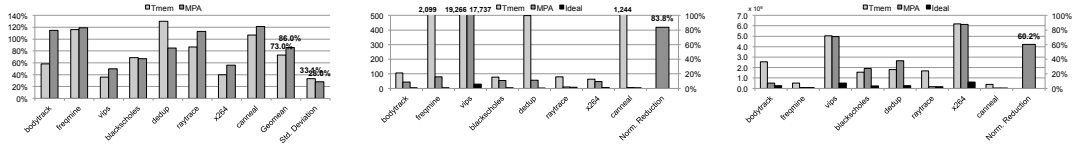
(a) Normalized performance to Ideal (b) Number of major page faults (c) Number of file reads

Figure IV.8: Performance analysis in 20% pressure



(a) Normalized performance to Ideal (b) Number of major page faults (c) Number of file reads

Figure IV.9: Performance analysis in 60% pressure



(a) Normalized performance to Ideal (b) Number of major page faults (c) Number of file reads

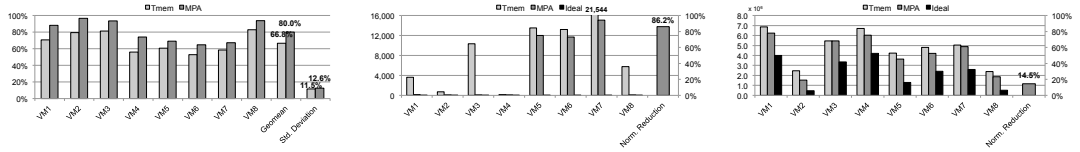
Figure IV.10: Performance analysis in 60% pressure

they run faster than MPA and Ideal. This is because *freqmine* and *dedup* are frequently rescheduled than other applications which wait for file I/O. Thus, with Tmem, *freqmine* and *dedup* steal scheduled slices from other applications because of those applications' increased file I/O. As a result we see higher performance degradation and standard deviation with Tmem, also bringing into question scheduling fairness. Tmem exhibits a standard deviation of 23.4% while MPA shows only 6.9%. Overall, the system wide major page faults decrease by 92.5% and the number of file reads decreases by 81.3% compared to Tmem.

In the 40% and 60% memory restriction experiments, MPA ballooning outperforms Tmem by 15.3% and 13.0% respectively. Since there is not enough memory to allocate, Tmem stores most of evicted anonymous pages and file pages in the hypervisor-level cache. MPA ballooning allows VMs to keep their actively used pages, avoiding page faults. As a result, MPA ballooning reduces page faults and file reads by 76.0% and 65.0% respectively under the 40% restriction model (Figure IV.9b and IV.9c). Similarly, the numbers decreases by 83.8% and 60.2% under the 60% restriction model (Figure IV.10b and IV.10c). In speedup, Tmem still shows outliers in both memory restriction models, with higher performance degradation and standard deviation compared to MPA ballooning. This result clearly shows that traditional ballooning technique does not work efficiently as the memory pressure increases.

IV.4.3 Multiple Applications in Random Order

In this experiment, we assume a scenario in which different applications are run in each VM. Based on Table IV.3, we assign multiple applications to each VM and execute them in random order. The experiment is stopped when all applications have repeated more than three instances. Due to the limited space, we show the experiment results from the worst and most realistic scenario (60% memory pressure) where the system experiences



(a) Normalized performance to Ideal (b) Number of major page faults (c) Number of file reads

Figure IV.11: Performance analysis with random ordered applications

transitions from *Low, Mild* and *Heavy pressure* regimes. The performance improvement in random ordered applications shows more consistent trend than in the single application repetition scenario. The MPA ballooning provides adaptive memory cushion so that the guest VMs can avoid major page faults and maintain file pages locally. In particular, the reallocation trigger and adaptive hysteresis become more important when multiple applications are mixed. The trigger increases the responsiveness of each VM when there is an urgent memory demand. At the same time, the adaptive hysteresis controls VMs with a large amount of inactive pages to return their pages quickly for other VMs with urgent needs. Combining these techniques together, as shown in Figure IV.11a, MPA ballooning achieves a 13.2% geomean speedup compared to Tmem. The number of major page faults dramatically decreases by 86.2% (Figure IV.11b) while the number of file I/Os decreases by 14.5% (Figure IV.11c). As a result, MPA ballooning reaches 80% performance of non-memory constrained system. Note that MPA ballooning always shows consistent performance improvement with any guest VM in a highly memory constrained model.

IV.4.4 Hypercall Overheads

Since MPA ballooning requires additional hypercalls to report the number of different types of pages from guest OS to hypervisor, one might be concerned that it results in greater hypercall overhead. Figure IV.12 measures the number of hypercalls from the ran-

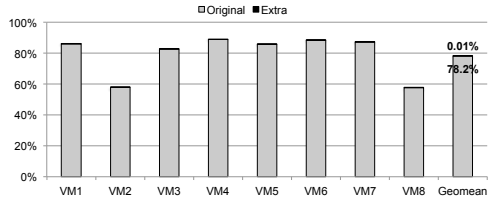


Figure IV.12: Number of hypercalls normalized to Tmem

domly mixed applications experiment (Section IV.4.3). In the figure, "Original" represents the fraction of hypercalls used by MPA to control page movement and change page table entries (similar to those used by Tmem). "Extra" represents the fraction of new hypercalls in MPA ballooning used to report malloc'ed/file pages and activate memory reallocation trigger. Since MPA ballooning leaves actively used pages in the guest VMs, it reduces useless page movement from VMs to the hypervisor. As a result, MPA ballooning greatly reduces the number of "Original" hypercalls, by 21.8% on average. Furthermore, the additional "Extra" hypercalls are less than 0.01% on average relative to Tmem's hypercalls. Thus, Figure IV.12 clearly shows *MPA, in fact, significantly reduces the overall number of hypercalls* relative to Tmem.

IV.5 Summary

In this chapter, we propose MPA ballooning, a VM memory management technique which dynamically adapts to the system memory pressure state. Prior works in VM memory management are oblivious to the system memory pressure and thus incur substantial performance overheads under different memory pressure regimes. We classify memory pressure into the *Low, Mild,* and *Heavy pressure* regimes based on the committed memory and file page usage. MPA ballooning leverages back-channel information from the guest VMs to dynamically allocate memory resources to each VM based on the current memory pressure regime. Moreover, the MPA ballooning proactively reacts and adapts to sudden changes in memory demand from guest VMs. To the best of our knowledge, MPA ballooning is the first VM memory management technique which dynamically changes the memory allocation policy based on the system's memory pressure regime. We show that MPA ballooning substantially improves performance versus baseline ballooning, regardless of memory pressure regime.

CHAPTER V

CONCLUSION

Due to the strict power and energy constraints on microprocessor design, it is impractical to naively increase the number of cores or the size of cache structures. Prior study [3] shows that regardless of chip organization and topology, multi-core scaling is power limited and does not achieve the projected performance gain. Under this severe restriction, intelligent memory management skills require minimal hardware or software complexity without losing significant performance benefits. Therefore, in modern microprocessor design, speculation-based memory management becomes an essential part to attain high-performance because its performance cost is far cheaper than increasing the actual memory capacity.

In this dissertation, we discussed three intelligent speculation mechanisms that efficiently manage the memory usage. First, we introduce SPP: a path-confidence data prefetcher covering a wide range of memory access patterns with prefetching confidence. Since each prefetch request in SPP holds a unique confidence value, it allows prefetcher to dynamically control the aggressiveness of speculation. Second, we present KPC: a holistic memory management for multi-level cache hierarchy. Unlike traditional LLC management techniques, KPC exploits the prefetching confidence as a mean of data placement and promotion in the LLC. In doing so, KPC integrates prefetching and replacement policy as a one unified speculation mechanism and provides superior performance compared to existing LLC management schemes. Third, we propose MPA: an efficient memory reallocation mechanism in virtualized system. MPA dynamically allocates memory across multiple virtual machines based on the total global pressure experienced by the main host system.

The aforementioned speculation techniques can be easily adopted by existing infras-

tructure without major modifications. For example, both SPP and KPC do not rely on the PC register which removes the complex logic between processor and on-chip caches. Also, MPA balooning requires less than 500 lines of code in Linux kernel and Xen hypervisor. The principle of minimal design complexity allows hardware/software engineers to comfortably work on the proposed ideas. As we emphasized in this dissertation, we certainly believe that an advanced prediction mechanism for future memory usage is an important feature of memory system designs and we expect more opportunities in this research area.

REFERENCES

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comp. Arch. News*, vol. 23, pp. 20–24, 1995.

[2] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. Leblanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, 1974.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, ACM, 2011.

[4] A. J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, pp. 7–21, 1978.

[5] T. Chen and J. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, pp. 609–623, 1995.

[6] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *Proceedings of the 33th Annual International Symposium on Computer Architecture (ISCA)*, pp. 252–263, IEEE Computer Society, 2006.

[7] I. Hur and C. Lin, "Memory prefetching using adaptive stream detection," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 397–408, IEEE Computer Society, 2006.

[8] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-temporal memory streaming," in *ISCA*, pp. 69–80, 2009.

[9] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching.," in *MICRO*, pp. 247–259, 2013.

[10] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for high performance data cache prefetch," *Journal of Instruction-Level Parallelism*, vol. 13, pp. 1–24, 2011.

[11] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015.

[12] P. Michaud, "A best-offset prefetcher," in *High Performance Computer Architecture (HPCA), 2016 IEEE 20th International Symposium on*, IEEE, 2016.

[13] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.

[14] S. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 175–186, IEEE, 2010.

[15] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 381–391, ACM, 2007.

[16] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 60–71, ACM, 2010.

[17] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer, "Ship: Signature-based hit predictor for high performance caching," in *Proceed-*

ings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 430–441, ACM, 2011.

[18] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 355–366, ACM, 2012.

[19] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," in *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*, pp. 176–186, IEEE, 1991.

[20] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 63–74, IEEE, 2007.

[21] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 623–634, IEEE Computer Society, 2014.

[22] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pp. 626–637, IEEE, 2014.

[23] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Microarchitecture (MICRO), 2016 49rd Annual IEEE/ACM International Symposium on*, IEEE, 2016.

[24] P. Michaud, "A best-offset prefetcher," in *High Performance Computer Architecture (HPCA), 2016 IEEE 20th International Symposium on*, IEEE, 2016.

[25] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jiménez, "Improving cache performance by exploiting read-write disparity," in *Proceedings of the 20th Internatial Symposiym on High Performance Computer Architecture (HPCA)*, pp. 452–463, IEEE, 2014.

[26] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction & dead-block correlating prefetchers," in *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*, pp. 144–154, IEEE, 2001.

[27] E. Teran, Y. Tian, Z. Wang, D. A. Jim, *et al.*, "Minimal disturbance placement and promotion," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 201–211, IEEE, 2016.

[28] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely Jr, and J. Emer, "Pacman: prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 442–453, ACM, 2011.

[29] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 267–278, ACM, 2012.

[30] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, p. 51, 2015.

[31] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS Operating Systems Review (SOSP)*, vol. 37, pp. 164–177, 2003.

[32] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 129–140, 2003.

[33] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "Ac/dc: An adaptive data cache prefetcher," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pp. 135–145, IEEE Computer Society, 2004.

[34] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Software, IEE Proceedings-*, pp. 96–96, IEEE, 2004.

[35] S. H. Pugsley, A. R. Alameldeen, C. Wilkerson, and H. Kim, "The 2nd Data Prefetching Championship (DPC-2)."

[36] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, pp. 318–319, ACM, 2003.

[37] "Standard Performance Evaluation Corporation CPU2006 Benchmark Suite.."

[38] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGPLAN Notices*, vol. 47, pp. 37–48, ACM, 2012.

[39] Y. Ishii, M. Inaba, and K. Hiraki, "Unified memory optimizing architecture: memory subsystem control with a unified predictor," in *Proceedings of the 26th ACM*

International Conference on Supercomputing, pp. 267–278, ACM, 2012.

[40] N. D. Enright Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: understanding the effects of multiprocessor prefetches," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 177–188, 2006.

[41] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*, p. 53, ACM, 2015.

[42] J.-Y. Won, P. Gratz, S. Shakkottai, and J. Hu, "Having your cake and eating it too: Energy savings without performance loss through resource sharing driven power management," in *Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on*, pp. 255–260, IEEE, 2015.

[43] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 284–296, ACM, 2013.

[44] V. V. Fedorov, S. Qiu, A. Reddy, and P. V. Gratz, "Ari: Adaptive llc-memory traffic management," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, p. 46, 2013.

[45] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, (Los Alamitos, CA, USA), pp. 222–233, IEEE Computer Society, 2008.

[46] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[47] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Transactions on Interactive Intelligent Systems (TiiS)*, vol. 5, no. 4, p. 19, 2016.

[48] R. Hegde, "Optimizing application performance on intel core microarchitecture using hardware-implemented prefetchers," *Intel Software Network*, 2008.

[49] D. Magenheimer, C. Mason, D. McCracken, and K. Hackel, "Transcendent memory and linux," in *Proceedings of the Ottawa Linux Symposium (OLS)*, pp. 191–200, Linux Symposium, 2009.

[50] C. A. Waldspurger, "Memory resource management in vmware esx server," in *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 181–194, USENIX Association, 2002.

[51] VMware, "Understanding memory resource management in vmware vsphere 5.0," in *Technical White Paper*, pp. 1–19, VMware, 2011.

[52] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, p. 1, USENIX Association, 1994.

[53] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *ACM SIGOPS Operating Systems Review*, vol. 38, pp. 177–188, ACM, 2004.

[54] P. Lu and K. Shen, "Virtual machine memory access tracing with hypervisor exclusive cache.," in *Usenix Annual Technical Conference (ATC)*, pp. 29–43, USENIX Association, 2007.

[55] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 3, pp. 37–47, 2009.

[56] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: a mechanism for speculative address translation," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pp. 307–317, IEEE, 2011.

[57] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Complication Techniques (PACT)*, pp. 72–81, ACM, 2008.

[58] A. Gordon, M. Hines, D. Da Silva, M. Ben-Yehuda, M. Silva, and G. Lizarraga, "Ginkgo: Automated, application-driven memory overcommitment for cloud computing," *Proc. RESoLVE*, 2011.

[59] D. Magenheimer, "Add self-ballooning to balloon driver," *Discussion on Xen Development mailing list and personal communication*, 2008.

[60] Intel, "Intel 64 and ia-32 architectures software developer's manual," vol. 3A, pp. 125–136, 2014.

[61] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. " O'Reilly Media, Inc.", 2005.

[62] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[63] M. Lee, A. S. Krishnakumar, P. Krishnan, N. Singh, and S. Yajnik, "Supporting soft real-time tasks in the Xen hypervisor," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS International conference on Virtual Execution Environments (VEE)*, pp. 97–108, ACM, 2010.