Department of Computer Architecture
Universitat Politècnica de Catalunya

# Advanced Hardware Prefetching in Virtual Memory Systems

A dissertation submitted in fulfillment of
the requirements for the degree of
*Doctor of Philosophy* in Computer Science

*Advisors: Marc Casas, Lluc Alvarez*

**Georgios Vavouliotis**

**2022**

Ἓν οἶδα ὅτι οὐδὲν οἶδα
— Σωκράτης

# Statement of Originality

I hereby declare that this thesis embodies scientific results of my own original research that have been published in international peer-reviewed conferences and journals, has been composed by myself, and has been checked by my supervisors before presentation. I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

**Georgios Vavouliotis,**

**October 2022**

# Acknowledgements

I would like to start by expressing my sincere gratitude to my advisors Marc Casas and Lluc Alvarez for their guidance and support throughout the conduction of this doctoral dissertation. Both have significantly contributed to my professional growth by giving me the opportunity to work on what I was interested in while always offering endless help and constructive criticism in discussing research ideas.

I also had the privilege to work closely with four exceptional collaborators: Prof. Vasileios Karakostas from the National and Kapodistrian University of Athens, Prof. Daniel A. Jimenez from the Texas A&M University, Prof. Boris Grot from the University of Edinburgh, and Prof. Paul V. Gratz from the Texas A&M University. Their contributions to my dissertation have been invaluable. I am also profoundly grateful to them for their advice that helped me transition from a student to a researcher. In particular, I feel indebted to Daniel for the many technical discussions during and after his sabbatical in Barcelona as well as his advice on how to communicate research ideas and write rebuttals. Finally, I want to thank Vasileios who was my co-advisor during my undergraduate studies at the National Technical University of Athens for having the patience to work with me during my first steps and for motivating me to pursue a doctoral program.

Special thanks to Prof. Dionisios Pnevmatikatos, Prof. Alberto Ros, and Dr. Xavier Martorell for serving on my thesis final defense committee and providing constructive feedback that improved the quality of this dissertation. I am also grateful to Dr. Oscar Palomar and Dr. Jaume Abella Ferrer for serving on my thesis pre-defense committee and offering thoughtful comments and advice.

I would also like to thank all the co-authors of the publications of this dissertation: Prof. Nectarios Koziris, Dr. Konstantinos Nikas, and Gino Chancon. Special thanks go to Gino for helping me with the last contribution of my thesis, and the numerous discussions we had about research, life, and coffee.

I am also thankful to Prof. Abhishek Bhattacharjee and Prof. Dionisios Pnevmatikatos for their constructive feedback on the first contribution of this dissertation. Special thanks go to Prof. Abhishek Bhattacharjee for his advice which was vital for publishing this work.

I am also grateful to my colleagues from Barcelona Supercomputing Center for their support and help during these years. Many thanks go to Vatistas Kostalampros, Dr. Dimitrios Chas-

apis, Alexandre Valentin Jamet, Dr. Miguel Moreto, Luc Jaumes, Isaac Sanchez, Vladimir Dimic, Asaf Badouh, Dr. Santiago Marco Sola, Francesc Martinez, Ruben Langarita Benitez, Robin Kumar Sharma, Guillem López Paradís, and Victor Soria Pardos. Special thanks go to my dear friend and colleague Vatistas for the support, help, and amazing memories from Barcelona, Greece, and Portugal. Finally, I thank Isabel Garcia who used to work for the HR of the Barcelona Supercomputing Center for her patience in helping me prepare the documents for the scholarship applications.

Next, I would like to thank my friends from Greece for the unforgettable moments. Specifically, I would like to thank Christos, Tasos, Vaso, Menelaos, Dimitris, Eleni, and Konstantina. I sincerely thank them all for always being there for me; each one with his unique style. Special thanks go to Christos for always supporting me, Tasos for the amazing campings and beer festivals we had on our island Evoia, Vaso and Menelaos for the great conversations we had when I was in Greece, Dimitris for the numerous calls we had over the years and the great vacation in Crete, Eleni for her unconditional support and the great haircuts, and Konstantina for reminding me that there are people who complain more than I do.

Last but certainly not least, I would like to express my deepest gratitude to my family for supporting me during all these years. First, to the most creative person I have ever met, my mother Eleni, the most hard-working and mentally strong person I know, my father Spyros, and the kindest person on earth, my sister Katerina, for their unconditional love. They have been supporting me in all possible matters, letting me discover my own life path. Finally, I would like to thank my grandmother Katerina (Katina) for her love and the amazingly funny moments she offers me; Roukoutangkous is definitely a word I will remember forever. This dissertation would not have been possible without them.

# Abstract

Despite groundbreaking technological innovations, the disparity between processor and memory speeds (known as Memory Wall) is still a major performance obstacle for modern systems. Hardware prefetching is a latency-tolerance technique that has proven successful at shrinking this bottleneck. Nearly all real-world $\mu$architectural designs employ various prefetchers. Consequently, hardware prefetching attracts a lot of research attention.

Virtual memory has been vital for the success of computing due to its programmability and security benefits. However, virtual memory does not come for free since each memory access requires a translation from the virtual to the physical address space that incurs high latency and energy overheads. To alleviate these overheads, a hardware cache, named Translation Lookaside Buffer (TLB), is typically employed to store the most recently used translations. However, TLBs are limited in capacity, thus there are not adequate for assuring high performance. Processor vendors address the need for fast address translation by providing dedicated support for virtual memory (*e.g.*, multi-level TLBs, multiple page sizes). Despite the existence of such support, the advent of applications with large data and code footprints aggravates the pressure placed on the virtual memory subsystem, resulting in frequent page walks that deteriorate system performance.

This dissertation argues that hardware prefetching can attenuate the Memory Wall bottleneck in virtual memory systems. To support our claim, we design and propose fully-legacy preserving TLB prefetching schemes and exploit address translation metadata that are available at the $\mu$architecture to improve the effectiveness of the prefetchers operating in the physical address space.

To reduce the overheads of frequent TLB misses due to data accesses, we propose a solution that consists of the *Sampling-Based Free TLB Prefetching (SBFP)* scheme and the *Agile TLB Prefetcher (ATP)*. SBFP exploits the locality in the last level of the page table to enhance the performance of TLB prefetching. ATP combines three prefetch engines while disabling TLB prefetching during phases that does not provide benefits. Across different benchmark suites, we show that ATP combined with SBFP improves performance over the best-performing prior TLB prefetcher while reducing the page walk references to the memory hierarchy.

Next, we argue that instruction address translation is an emerging bottleneck in servers. To support our claim, we characterize the TLB behavior of server workloads and provide evidence that instruction address translation is a bottleneck in servers. To attenuate this

bottleneck, we propose *Morrigan*, the first instruction TLB prefetcher. Morrigan combines a sequential prefetcher with an ensemble of hardware Markov prefetchers that build variable length Markov chains out of the instruction TLB miss stream while using a new frequency-based replacement policy. Across a set of industrial server workloads, Morrigan provides great performance gains while eliminating the majority of the demand page walks for instruction accesses.

Our last contribution improves the efficacy of cache prefetchers operating in the physical address space by exploiting modern support for large pages. We propose the *Page-size Propagation Module (PPM)*, a $\mu$architectural scheme that transmits the page size information to the lower-level cache prefetchers and enables safe prefetching beyond 4KB physical page boundaries. We further design a module comprised of two prefetchers that both exploit PPM but drive prefetching decisions assuming different page sizes. Our experiments reveal that the proposed page size exploitation techniques provide great performance enhancements on various state-of-the-art cache prefetchers.

The proposals of this dissertation are fully legacy-preserving, do not call for disruptive changes, do not require any OS involvement, and constitute practical solutions to real-world bottlenecks.

# Extended Abstract

Despite groundbreaking technological innovations and revolutions, the discrepancy between processor and main memory speeds is still a major performance obstacle for modern systems and is widely known as the Memory Wall. Hardware prefetching is a latency-tolerance technique that aims at attenuating the Memory Wall bottleneck. The core idea behind hardware prefetching is to proactively fetch memory blocks into the on-chip caches before they are explicitly demanded by a core, giving the illusion to the application that memory accesses need a few cycles to complete. Hardware prefetching has proven successful at shrinking the processor-memory performance gap and nearly all real-world $\mu$architectural designs employ various hardware prefetchers for the different cache levels. However, state-of-the-art designs of hardware prefetchers are far from approaching the performance of an ideal prefetcher. As a result, hardware prefetching attracts a lot of research attention since it has the potential to provide outstanding performance and energy enhancements without disrupting the existing memory subsystem. Indeed, there are myriads of hardware prefetchers with different properties proposed in recent literature.

Virtual memory is a memory management technique that has been vital for the success of computing due to its unique programmability and security benefits. Nearly all modern computing systems today, from desktops to servers and datacenters, implement virtual memory while programmers do not even think about the existence of virtual memory when writing code today. However, virtual memory does not come for free. In virtual memory systems, memory accesses are performed using virtual addresses, thus a memory access requires a translation from the virtual address space to the physical address space that incurs high latency and energy overheads. To alleviate the address translation overheads, a hardware cache, named *Translation Lookaside Buffer (TLB)*, is typically employed to store the most recently used address translation entries. TLBs are placed close to the core to ensure fast address translation upon memory accesses. However, TLBs are limited in capacity, thus there are not adequate for assuring high-performance address translation. Leading processor vendors address the need for fast address translation by providing dedicated hardware and software support to improve the performance of the virtual memory subsystem; from multi-level TLB hierarchies to hardware page table walkers and multiple page sizes, among others. Despite the existence of sophisticated architectural support for address translation, the advent of applications with large data and code footprints aggravates the pressure

placed on the virtual memory subsystem, resulting in frequent page walks for both data and instruction accesses that deteriorate the performance of the system.

Virtual memory makes the Memory Wall 'taller' due to the requirement of traversing the memory hierarchy multiple times per TLB miss to obtain the requested address translation from the page table. To make matters worse, the advent of emerging workloads with massive data and code footprints that experience high TLB miss rates, place tremendous pressure on the memory hierarchy due to need for frequently performing page walks, threatening the performance of computing.

In this dissertation, we argue that sophisticated hardware prefetching has the potential to attenuate the Memory Wall bottleneck in virtual memory systems. In this direction, we (i) design and propose fully-legacy preserving hardware prefetching schemes for the TLB hierarchy that aim at reducing the TLB miss rates of both data and instruction references, and (ii) exploit address translation metadata that are available at the $\mu$architecture to improve the effectiveness of any hardware cache prefetcher operating in the physical address space without opening new security vulnerabilities.

To reduce the performance overheads of frequent TLB misses due to data accesses, we focus on hardware TLB prefetching because it is an approach that relies only on the memory access patterns of the application, is independent of the system state, and does not imply any OS involvement. We propose a composite solution that consists of the *Sampling-Based Free TLB Prefetching (SBFP)* scheme and the *Agile TLB Prefetcher (ATP)*. SBFP is a $\mu$architectural scheme that exploits the locality in the last level of the radix tree page table to enhance the performance of TLB prefetching while reducing the memory footprint of page walks and the energy consumption of address translation. ATP is a composite TLB prefetcher that efficiently combines three easily implementable prefetch engines by leveraging an adaptive selection scheme while disabling TLB prefetching during phases that does not provide benefits. Across a set of 150 workloads spanning different academic and industrial benchmark suites, we demonstrate that our proposal, ATP combined with SBFP, significantly improves geometric mean performance over the best-performing prior TLB prefetcher while significantly reducing the page walk references to the memory hierarchy and the dynamic energy consumption of address translation, at a cost of only 2KB of storage.

Next, we focus on the domain of address translation associated with instruction references where we argue that instruction address translation is an emerging performance bottleneck in servers and datacenters. To support our claim, we provide the first ever $\mu$architectural study that (i) characterizes the TLB behavior of industrial server workloads, and (ii) provides evidence that instruction address translation is a performance bottleneck in servers. To attenuate the instruction address translation bottleneck of big code applications, we design and propose *Morrigan*, the first ever hardware TLB prefetcher for instruction accesses. Morrigan is a fully legacy-preserving prefetcher that consists of two complementary prefetch engines capable of capturing both regular and irregular instruction TLB miss patterns: (i) the Irregular Instruction TLB Prefetcher (IRIP) which is an ensemble of hardware

Markov prefetchers that dynamically build variable length Markov chains out of the instruction TLB miss stream while using a novel frequency-based replacement policy, and (ii) the Small Delta Prefetcher (SDP) which is a simple sequential TLB prefetcher that is activated only when the IRIP module fails at producing prefetch requests. Across 45 industrial server workloads, Morrigan improves geometric mean performance by 7.6% while eliminating 69% of the demand page walks, at the cost of 3.7KB of storage.

Our last contribution improves the efficacy of cache prefetchers operating in the physical address space by exploiting modern prevalence and support for large pages, *i.e.*, page sizes larger than standard 4KB pages. We design and propose the *Page-size Propagation Module (PPM)*, a $\mu$architectural scheme that efficiently propagates the page size information to the lower-level cache prefetchers and enables safe prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page, at the cost of augmenting the MSHRs of the first-level caches with one bit. In addition, we demonstrate that PPM does not introduce security vulnerabilities. To capitalize on PPM's benefits, we design a composite module comprised of two cache prefetchers that both exploit PPM but drive prefetching decisions assuming different page sizes. This composite prefetching module uses adaptive selection logic to dynamically switch between the two page size aware prefetchers and is transparent to which cache prefetcher is used. Our experimental campaign reveals that the proposed page size exploitation techniques provide significant performance enhancements on various state-of-the-art lower-level cache prefetchers (up to 8.1% geometric mean speedup over the original version of the prefetchers) while not harming the performance of non memory-intensive workloads, at modest storage overheads.

This dissertation demonstrates that (i) hardware TLB prefetching is a promising solution for the address translation bottleneck for both data and instruction references, (ii) instruction address translation is an emerging problem in servers, and (iii) exploiting the presence of large pages in modern systems for improving the performance of hardware cache prefetching provides significant benefits. All the proposals of this dissertation are fully legacy-preserving, do not call for disruptive changes, do not require any OS involvement, and constitute practical solutions to real-world bottlenecks since they incur minimal storage overheads. Consequently, they have great potential to influence future industrial designs and initiate additional research on hardware prefetching for virtual memory systems.

# Thesis Organization

This doctoral dissertation is divided into eight chapters following the subsequent structure.

**Chapter 1** introduces the Memory Wall bottleneck, the fundamental idea behind hardware prefetching, and basic as well as advanced concepts of virtual memory. Then, it motivates the need for improving hardware prefetching for virtual memory systems and presents the main contributions of this doctoral thesis.

**Chapter 2** provides additional background on Memory Wall, hardware prefetching, and architectural support for address translation. Furthermore, it provides all the necessary information about hardware prefetching applied for the virtual memory subsystem while presenting important features that virtual memory brings to hardware cache prefetching.

**Chapter 3** presents the *Agile TLB Prefetcher (ATP)* and the *Sampling-Based Free TLB Prefetching (SBFP)*, two μarchitectural modules capable of accelerating address translation associated with data accesses. This chapter is based on our work published in the 48th International Symposium on Computer Architecture (ISCA 2021) [284].

**Chapter 4** presents *Morrigan*, the first ever μarchitectural TLB prefetcher for instruction references that targets server applications with big code footprints. This chapter is based on our work published in the 54th International Symposium on Microarchitecture (MICRO 2021) [283].

**Chapter 5** presents *Page Size Aware Cache Prefetching*, the first work that enables safe cache prefetching beyond 4KB physical page boundaries while revealing how address translation metadata could be effectively leveraged for improving cache prefetching performance. This chapter is based on our work published in the 55th International Symposium on Microarchitecture (MICRO 2022) [285].

**Chapters 3, 4, and 5** also suggest various future research directions in their domain.

**Chapter 6** summarizes the contributions of this doctoral thesis while discussing its broader impact and acknowledging its main supporters.

**Chapter** 7 presents a list of publications that have been accepted in peer-reviewed conferences and journals as well as ongoing works and collaborations.

**Chapter** 8 illustrates the author's vision for future research.

# Contents

**Statement of Originality** v

**Acknowledgements** vii

**Abstract** ix

**Extended Abstract** xi

**Thesis Organization** xv

**Contents** xvii

**List of Figures** xxiii

**List of Tables** xxxiii

**1 Introduction** 1

1.1 Memory Wall . . . . . 1
1.2 Hardware Prefetching: Endless Opportunity Domain . . . . . 2
1.3 Virtual Memory Systems . . . . . 4
1.3.1 Basic Architectural Support for Address Translation . . . . . 4
1.3.2 Advanced Hardware Support for Address Translation . . . . . 5
1.4 Motivation . . . . . 7
1.4.1 Mismatch in TLB and Memory Sizes . . . . . 7
1.4.2 Unexploited Address Translation Metadata . . . . . 8
1.5 Thesis Approach . . . . . 8

1.6 Thesis Contributions . . . . . 10
1.6.1 Agile Prefetching for the Data TLB Miss Stream . . . . . 10
1.6.2 Markov-based Prefetching for the Instruction TLB Miss Stream . . . . 12
1.6.3 Safe Cache Prefetching Beyond 4KB Page Boundaries . . . . . 14

**2 Background** **17**
2.1 Memory Wall . . . . . 18
2.2 Hardware Prefetching . . . . . 21
2.2.1 Oracle Hardware Prefetcher . . . . . 22
2.2.2 $W^3$ Challenge . . . . . 23
2.2.3 Efficacy Metrics . . . . . 24
2.2.4 Classification of Prior Work . . . . . 26
2.2.4.1 Spatial Prefetching . . . . . 27
2.2.4.2 Temporal Prefetching . . . . . 27
2.2.5 Cache Prefetchers in Modern Chips . . . . . 28
2.3 Virtual Memory . . . . . 30
2.3.1 Virtual Memory Implementation . . . . . 31
2.3.2 Building Blocks of Page-based Virtual Memory . . . . . 32
2.3.3 Architectural Support for Address Translation . . . . . 34
2.3.3.1 Page Table . . . . . 36
2.3.3.2 Translation Lookaside Buffer (TLB) . . . . . 40
2.3.3.3 Hardware Page Table Walkers . . . . . 46
2.3.3.4 MMU-Caches . . . . . 48
2.3.3.5 Software Schemes and Policies . . . . . 49
2.4 Hardware Prefetching in Virtual Memory Systems . . . . . 50
2.4.1 Hardware Prefetching for the Virtual Memory Subsystem . . . . . 51
2.4.1.1 TLB Prefetching . . . . . 51
2.4.1.2 Previously Proposed TLB Prefetchers . . . . . 54
2.4.2 Cache Prefetching in Virtual Memory Systems . . . . . 58

**3 Agile Data TLB Prefetching** **61**
3.1 Introduction . . . . . 61
3.2 Background . . . . . 64

3.2.1 Page Table Locality . . . 64
3.3 Motivation . . . 66
3.3.1 Quantifying the Potential of TLB Prefetching . . . 69
3.3.2 Does Any Prior TLB Prefetcher Dominate? . . . 69
3.3.3 What is the Impact of TLB Prefetching on Page Walk References? . . . 70
3.3.4 What is the Potential of Exploiting Page Table Locality? . . . 71
3.3.5 Putting Everything Together . . . 72
3.4 Sampling-Based Free TLB Prefetching (SBFP) . . . 72
3.4.1 Pushing the Envelope on Free TLB Prefetching . . . 72
3.4.2 SBFP Design and Operation . . . 73
3.4.2.1 Design Overview . . . 73
3.4.2.2 SBFP Operation . . . 75
3.4.3 Combining SBFP with TLB Prefetching Schemes . . . 76
3.4.4 Discussion . . . 77
3.5 Agile TLB Prefetcher (ATP) . . . 78
3.5.1 Design Overview . . . 78
3.5.2 ATP Operation . . . 79
3.5.3 Building Blocks of ATP . . . 80
3.5.4 Discussion . . . 83
3.6 ATP+SBFP : Additional Considerations . . . 84
3.7 Methodology . . . 85
3.7.1 Simulation Infrastructure . . . 85
3.7.2 Evaluated TLB Prefetchers . . . 86
3.7.3 Policies Exploiting PTE Locality . . . 87
3.7.4 Workloads . . . 88
3.8 Experimental Campaign . . . 90
3.8.1 Impact of SBFP . . . 90
3.8.1.1 TLB Coverage and Performance . . . 90
3.8.1.2 Cost of TLB prefetching . . . 94
3.8.2 Evaluation of ATP coupled with SBFP . . . 96
3.8.2.1 Performance Comparison . . . 96
3.8.2.2 Cost of TLB Prefetching . . . 100
3.8.2.3 Storage Budget . . . 102

3.8.2.4 Large Pages . . . 103
3.8.2.5 Energy Consumption . . . 104
3.8.3 Comparison with Other Approaches . . . 105
3.8.4 Interaction with OS Page Replacement Policy . . . 108
3.9 Related Work . . . 109
3.10 Summary . . . 111
3.11 Future Work . . . 111

**4 Markov-based Instruction TLB Prefetching** **113**
4.1 Introduction . . . 113
4.2 Background . . . 117
4.3 Motivation . . . 118
4.3.1 Front-end Bottleneck . . . 118
4.3.2 Analyzing Industrial Server Workloads . . . 119
4.3.3 Understanding the Instruction TLB Misses . . . 121
4.3.4 Can Existing data TLB Prefetchers Help? . . . 124
4.3.5 Instruction Cache Prefetching . . . 127
4.3.6 Putting Everything Together . . . 128
4.4 Morrigan . . . 129
4.4.1 Design . . . 129
4.4.1.1 Irregular Instruction TLB Prefetcher (IRIP) . . . 129
4.4.1.2 Small Delta Prefetcher (SDP) . . . 134
4.4.2 Operation of Morrigan . . . 134
4.4.3 Additional Aspects . . . 137
4.5 Methodology . . . 139
4.5.1 Simulation Infrastructure . . . 139
4.5.1.1 Simulated Page Sizes . . . 140
4.5.2 Evaluated TLB Prefetchers . . . 141
4.5.3 Workloads . . . 141
4.6 Experimental Campaign . . . 143
4.6.1 IRIP Module . . . 143
4.6.1.1 Miss Coverage . . . 144
4.6.1.2 Replacement Policy . . . 144

4.6.1.3 Configuring IRIP . . . . . . . . . . . . . . . . . . . . . . . . . . . . 145
4.6.2 Comparison with Data TLB Prefetchers . . . . . . . . . . . . . . . . . . . 146
4.6.3 Comparing Different IRIP Designs . . . . . . . . . . . . . . . . . . . . . . 148
4.6.4 Comparison with Other Approaches . . . . . . . . . . . . . . . . . . . . . 149
4.6.5 Synergy with L1i Cache Prefetching . . . . . . . . . . . . . . . . . . . . . 151
4.6.6 Workload Colocation in SMT Cores . . . . . . . . . . . . . . . . . . . . . 152
4.7 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
4.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156
4.9 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 156

**5 Page Size Aware Cache Prefetching** **159**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 159
5.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 165
5.2.0.1 Large Pages in Practice . . . . . . . . . . . . . . . . . . . . . . . . . 165
5.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
5.3.1 Limitations of Existing Cache Prefetchers . . . . . . . . . . . . . . . . . . 167
5.3.2 Opportunity for Safe Prefetching Across 4KB Boundaries . . . . . . . . 167
5.3.2.1 Quantifying the Potential . . . . . . . . . . . . . . . . . . . . . . . 169
5.3.3 Integrating Large Pages into the Design . . . . . . . . . . . . . . . . . . . 171
5.3.4 Putting Everything Together . . . . . . . . . . . . . . . . . . . . . . . . . 173
5.4 Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
5.4.1 Page-size Propagation Module (PPM) . . . . . . . . . . . . . . . . . . . . 174
5.4.1.1 Implementation and Operation . . . . . . . . . . . . . . . . . . . . 175
5.4.1.2 Additional Aspects . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
5.4.2 Integrating Large Pages in the Design . . . . . . . . . . . . . . . . . . . . 178
5.4.2.1 Design of Pref-PSA-2MB . . . . . . . . . . . . . . . . . . . . . . . 179
5.4.2.2 Selection Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 179
5.4.2.3 Pref-PSA-SD Operation . . . . . . . . . . . . . . . . . . . . . . . . 180
5.5 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
5.5.1 Performance Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
5.5.1.1 Constrained Evaluation . . . . . . . . . . . . . . . . . . . . . . . . 182
5.5.2 Workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
5.5.3 Multi-Core Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184

5.5.4 Evaluated Prefetchers . . . . . 184
5.6 Experimental Campaign . . . . . 185
5.6.1 Single Core Experiments . . . . . 185
5.6.1.1 Performance . . . . . 185
5.6.1.2 Sources of Performance Enhancements . . . . . 190
5.6.1.3 Different Selection Logic Implementations . . . . . 194
5.6.1.4 Constrained Evaluation . . . . . 195
5.6.1.5 Comparison with L1d Prefetching . . . . . 196
5.6.2 Multi-Core Experiments . . . . . 197
5.7 Summary . . . . . 199
5.8 Future Work . . . . . 199

**6 Conclusions** **201**
6.1 Additional Future Work . . . . . 203
6.2 Additional Acknowledgements . . . . . 203

**7 Publications** **205**
7.1 Conference Publications . . . . . 205
7.2 Other Publications . . . . . 206

**8 Research Vision** **209**
8.1 Software-Assisted $\mu$architectural Designs . . . . . 211
8.2 ML for $\mu$architectural Prediction and Prefetching . . . . . 211
8.3 Other Domains . . . . . 211
8.4 Industry, Academia, and Teaching . . . . . 212

**Abbreviations** **213**

**Bibliography** **217**

# List of Figures

1.1 Cartoon illustrating the contributions of this thesis integrated into a modern $\mu$architecture. The first contribution (blue) is a composite TLB prefetcher for data accesses. The second contribution (orange) is the first ever TLB prefetcher for instruction accesses. The third contribution (green) is the first work to exploit address translation metadata available at the $\mu$architecture for improving the efficacy of lower-level cache prefetchers. . . . . . . . . . . 9

2.1 Disparity between processor and main memory speeds for the last 25 years. . 18
2.2 Hierarchy of cache memories. Caches can be placed either on-chip or off-chip. 19
2.3 Anatomy of a modern cache hierarchy composed of three cache levels. The latency and capacity parameters are extracted from a recent Intel Icelake chip [29]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
2.4 Operation of a generic hardware prefetcher (PF) placed alongside a cache memory. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
2.5 Prefetching can be applied at any level of a modern cache hierarchy. PF refers to a generic hardware cache prefetcher. . . . . . . . . . . . . . . . . . . . . . . . 22
2.6 Simplistic execution diagram of a system with one-level cache hierarchy equipped (A) without any hardware prefetcher, (B) with the oracle hardware prefetcher, and (C) with a real-world hardware prefetcher. Prefetch requests are classified into on-time, late, early, and inaccurate. . . . . . . . . . 23
2.7 The fundamental idea behind virtual memory systems → each memory access requires a translation from the virtual address space to the physical address space. Dotted lines indicate actions. . . . . . . . . . . . . . . . . . . . . . . . 30

2.8 Cartoon depicting the abstraction layer of page-based virtual memory. Blue and orange boxes represent virtual and physical pages, respectively. Arrows representing virtual to physical mappings are one-to-one for visibility; this is not necessarily the case due to *homonyms* and *synonyms* [87]. Homonyms refer to cases where on virtual address points to multiple physical addresses. Synonyms refer to cases where multiple virtual addresses point to a single physical address. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
2.9 Conventional (top) and modern (bottom) software and hardware support for address translation. Modern TLBs have two levels; first-level TLBs are split between data and instructions (dTLB and iTLB, respectively) while L2 TLBs accommodate both data and instruction address translations. Modern designs further split TLBs between different page sizes. MMU-Caches are small cache structures that store entries of the different page table levels. Page table walkers are hardware finite state machines (FSMs) that perform page walks entirely in hardware. Private and shared caches are highlighted in white color since they are not structures dedicated to address translation, but do accommodate PTEs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.10 Illustration of the address translation process, containing more details than Figure 2.7. Diamonds indicate decision points while dotted lines indicate actions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
2.11 PTE format on x86-64 architectures assuming standard 4KB pages. . . . . . . 37
2.12 Radix tree page table with 4 levels and 48-bit canonical virtual addresses. The names of the page table levels (from left to right) are *Page Map Level 4 (PML4)*, *Page Directory Pointer (PDP)*, *Page Directory (PD)*, and *Page Table (PT)*. Moving to 57-bit canonical virtual addresses would require implementing a radix tree page table with 5 levels. . . . . . . . . . . . . . . . . . . . . . . . 38
2.13 (A) Format of a PDP entry that maps to a 1GB page (up) and references a PD (down). (B) Format of a PD entry that maps to a 2MB page (up) and references a PT (down). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
2.14 The process of address translation when a system employs a generic TLB that stores the most recently used PTEs. Diamonds indicate decision points while dotted lines indicate actions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

2.15 TLB hierarchy with two levels (left). L1 TLBs are split between instructions and data. L2 TLBs are larger and slower than L1 TLBs and store both data and instruction PTEs. The flow diagram (right) depicts the process of address translation when a system employs a 2-level TLB hierarchy. Diamonds indicate decision points while dotted lines indicate actions. . . . . . . . . . . 42
2.16 TLB organization in modern x86-64 systems that support 4KB, 2MB, and 1GB pages. The capacity of the different TLB structures is taken from Intel's Skylake 2018 chips [48]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
2.17 TLB placement relative to first-level hardware caches. There are three realistic configurations: (a) Physically Indexed, Physically Tagged (PIPT) caches, (b) Virtually Indexed, Physically Tagged (VIPT) caches, (c) Virtually Indexed, Virtually Tagged (VIVT) caches. Finally, Physically Indexed, Virtually Tagged (PIVT) caches are ignored because they are not used in practice. . . . . . . . . 45
2.18 The building blocks of a hardware page table walker (PTW) are (i) a finite state machine, and (ii) a buffer storing information about outstanding page walk references to the memory hierarchy (caches, DRAM). . . . . . . . . . . 46
2.19 MMU-Caches implemented in Intel x86-64 architectures. . . . . . . . . . . . . . 48
2.20 Cartoon depicting the $\mu$architecture of a system that employs a two-level TLB hierarchy, VIPT L1 caches and PIPT lower-level caches. . . . . . . . . . . . . . . 51
2.21 Prefetch Buffer (PB) entry. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
2.22 Operation of a system that employs a generic hardware TLB prefetcher. Diamonds indicate decision points while dotted lines indicate actions. . . . . . . 53
2.23 SP's basic operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
2.24 ASP's operation upon prediction table hits. Diamonds indicate decision points. 55
2.25 DP's operation upon prediction table hits. Diamonds indicate decision points. 56
2.26 MP's operation upon prediction table hits. Diamonds indicate decision points. 57

3.1 Locality in the last level of the radix tree page table in x86-64 architectures, assuming a 4-level radix tree page table and a TLB miss for virtual page 0xA3. This thesis interchangeably uses the terms *PTE locality* and *page table locality* to refer to this locality. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
3.2 Performance of SP, ASP, DP, and Perfect TLB with and without exploiting PTE locality, illustrated in Figure 3.1. Higher is better. . . . . . . . . . . . . . . . . . 67

3.3 Distribution of the normalized number of memory references due to page walks (demand and prefetch) depicted with box plots. Lower is better. . . . . 68
3.4 Illustration of the *free distance* concept, assuming a page walk for virtual page 0xA3. . . . . 73
3.5 Sampling-Based free TLB prefetching (SBFP) mechanism. The example considers a TLB miss for virtual page 0xA3, similar to Figure 3.4. The terms *vpn*, *ppn*, and *thr* refer to virtual page number, physical page number, and threshold, respectively. Diamonds indicate decision points. . . . . 74
3.6 Operation when SBFP is combined with a generic TLB prefetcher. Diamonds indicate decision points while dotted lines indicate actions. . . . . 76
3.7 Design of the Agile TLB Prefetcher (ATP). P0, P1, and P2 represent generic TLB prefetchers. C0, C1, and C2 are saturating counters. Disable refers to disabling of TLB prefetching. . . . . 78
3.8 Functionality flowchart showing the operation of ATP. Diamonds indicate decision points. . . . . 79
3.9 Operation of STP. . . . . 80
3.10 H2P's operation. . . . . 81
3.11 MASP's operation. . . . . 82
3.12 TLB MPKI reduction offered by different TLB prefetchers (SP, DP, ASP, STP, H2P, MASP, ATP) when combined with different scenarios that exploit free TLB prefetching (NoFP, NaiveFP, StaticFP, SBFP). Higher is better. The QMM, SPEC, and BD workloads experience 13.9, 3.4, and 38.9 average TLB MPKI in the baseline that does not apply TLB prefetching, respectively. . . . . 91
3.13 Geometric mean speedups delivered when different TLB prefetchers (SP, DP, ASP, STP, H2P, MASP, ATP) are combined with different scenarios that exploit free TLB prefetching (NoFP, NaiveFP, StaticFP, SBFP). Higher is better. . . . . 92
3.14 Normalized hit ratio of each free distance depicted with violin plots, across all considered workloads and TLB prefetchers. . . . . 93
3.15 Normalized memory references due to page walks (demand and prefetch) across all considered TLB prefetchers and scenarios that exploit page table locality. Lower is better. . . . . 95

3.16 Performance comparison between ATP coupled with SBFP and previously proposed TLB prefetchers (SP, DP, ASP). The baseline does not employ TLB prefetching. The evaluation of the QMM, BD, and SPEC workloads is presented in the top, middle, and bottom subfigures, respectively. Higher is better. 97
3.17 Performance comparison between ATP coupled with SBFP and previously proposed TLB prefetchers (SP, DP, ASP), considering the entire SPEC CPU 2006 and SPEC CPU 2017 benchmark suites. Higher is better. . . . . . . . . . 98
3.18 Fraction of TLB misses that ATP enables H2P, STP, MASP or disables TLB prefetching. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
3.19 Percentage of PB hits provided by ATP (its constituent prefetchers) and SBFP. 100
3.20 Distribution of the normalized number of memory references due to page walks (demand plus prefetch) depicted with violin plots (up). Breakdown of the average memory references caused by demand and prefetch page walks depending on which level of the memory hierarchy serves these references (down). The baseline memory references correspond to 100%. . . . . . . . . . 101
3.21 Distribution of speedups delivered by ATP coupled with SBFP and previously proposed TLB prefetchers (SP, DP, ASP) when the baseline system uses only 2MB pages and does not employ TLB prefetching. Higher is better. . . . . . . 103
3.22 Impact of ATP with SBFP and previously proposed TLB prefetchers (SP, DP, ASP) on the dynamic energy consumption of address translation depicted with box plots. Lower is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104
3.23 Performance comparison of ATP combined with SBFP with other approaches that improve TLB performance. Higher is better. . . . . . . . . . . . . . . . . . . . 106

4.1 Morrigan integrated into a modern $\mu$architecture. . . . . . . . . . . . . . . . . . . 116
4.2 Instruction TLB MPKI of Java server workloads from the Java DaCapo [89] and Java Renaissance [224] benchmark suites. . . . . . . . . . . . . . . . . . . . . 118
4.3 Average instruction MPKI (iMPKI) for front-end structures (L1i cache, iTLB, L2 TLB) across the QMM and SPEC suites. The L2 TLB iMPKI considers only the instruction L2 TLB misses. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 120
4.4 Cycles spent serving instruction TLB accesses across all QMM server workloads. 121

4.5 Accumulative distribution of deltas (absolute values) between pages that produce consecutive instruction TLB misses across all considered QMM server workloads. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
4.6 Instruction pages that produce at least one TLB miss, sorted by TLB miss frequency. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
4.7 Number of successor pages per instruction page that misses in the TLB, across all considered QMM server workloads. . . . . . . . . . . . . . . . . . . . . . . . . . 124
4.8 Probability of accessing the same successor page after an instruction TLB miss for a given instruction page, across all QMM server workloads. . . . . . . 124
4.9 Performance comparison between the previously proposed data TLB prefetchers (SP, ASP, DP, MP) when prefetching for the instruction TLB miss stream of the QMM server workloads and the ideal scenario where all instruction TLB accesses are hits (Perfect L2 TLB (inst)). Higher is better. . . . . . . . . . . . . 125
4.10 Performance of the FNL+MMA L1i cache prefetcher with and without taking into account instruction address translation, across all QMM server workloads. The baseline system utilizes the next-line L1i cache prefetcher. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
4.11 Design of the PRT-S2 prediction table. VPN, $D_{ij}$, and $C_{ij}$ refer to virtual page number, predicted distance, and confidence counter, respectively. . . . . . . . 130
4.12 Operation of the IRIP module on PRT-S2 hits. VPN, $D_{ij}$, and $C_{ij}$ refer to virtual page number, predicted distance, and confidence counter, respectively. Diamonds indicate decision points. . . . . . . . . . . . . . . . . . . . . . . . . . . . 132
4.13 Operation of the IRIP module on PRT-S2 misses. Diamonds indicate decision points. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
4.14 Complete operation of Morrigan in steps. Diamonds indicate decision points while dotted lines indicate actions. . . . . . . . . . . . . . . . . . . . . . . . . . . . 135
4.15 Miss coverage of Morrigan for various storage budgets. Higher is better. . . . 143
4.16 Miss coverage of Morrigan when the prediction tables of the IRIP module use different replacement policies across various storage budgets. Higher is better.145
4.17 Performance comparison between Morrigan and the prior data TLB prefetchers, presented in Section 2.4.1.2. All data TLB prefetchers have been configured to match the storage budget of Morrigan. The baseline system does not apply TLB prefetching. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . 146

4.18 Distribution of the normalized number of memory references due to demand and prefetch page walks depicted with violin plots. The baseline does not apply TLB prefetching. Lower is better. . . . . . . . . . . . . . . . . . . . . . . . . 147
4.19 Performance of Morrigan when the IRIP module uses an ensemble of four tables (Morrigan) versus a single-table design (Morrigan-mono). The baseline system does not apply TLB prefetching. Higher is better. . . . . . . . . . . . . . 149
4.20 Performance comparison of Morrigan with other state-of-the-art approaches aimed at improving TLB performance. The baseline system does not apply TLB prefetching. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . 150
4.21 Impact of Morrigan on the performance of L1i cache prefetching. The baseline system does not apply TLB prefetching and uses the next-line L1i cache prefetcher. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 152
4.22 Performance of Morrigan under SMT colocation across 50 randomly chosen pairs of QMM server workloads. The baseline system does not apply TLB prefetching. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153

5.1 Data structure residing on multiple 4KB pages (up) and one large page (down). Arrows (black and red) illustrate the memory access patterns. Red arrows represent patterns across 4KB pages that a cache prefetcher operating on the physical address space is not allowed to prefetch for (even if it correctly identifies the patterns). Note that there are no red arrows when the data structure resides in a large page. . . . . . . . . . . . . . . . . . . . . . . . . . . . 160
5.2 Probability distribution depicted with violin plots showing the probability for a given prefetch to be discarded because it attempts to cross 4KB physical page boundaries when the block resides in a large page, considering four state-of-the-art spatial cache prefetchers (SPP [172], VLDP [251], PPF [80], BOP [195]) and 80 memory-intensive applications, presented in Section 5.5. 162
5.3 Percentage of allocated memory mapped to 2MB large pages across the entire execution of nine representative memory-intensive benchmarks from the SPEC CPU 2006 benchmark suite [136], SPEC CPU 2017 benchmark suite [42], and GAP [74] benchmark suite, running on an Intel Xeon E5-2687W machine. Most workloads preserve the high usage of 2MB large pages during their entire execution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 168

5.4 Performance comparison between the original implementation of SPP and the ideal page size aware version of SPP (SPP-PSA-Magic) across the same set of memory-intensive workloads used for the physical machine measurements, presented in Figure 5.3. The baseline system does not apply prefetching at any cache level. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . 170

5.5 Performance comparison between the original implementation of SPP, the ideal page size aware version of SPP presented in Figure 5.4 (SPP-PSA-Magic), and deal page size aware SPP that inherently uses 2MB pages (SPP-PSA-Magic-2MB) across the same set of memory-intensive workloads used for the physical machine measurements, presented in Figure 5.3. The baseline system does not apply prefetching at any cache level. Higher is better. . . . . 172

5.6 Operation of a system that employs the Page-size Propagation Module (PPM). Diamonds indicate decision points. . . . . . . . . . . . . . . . . . . . . . . . . . . 175

5.7 Operation of a system that leverages PPM for propagating the page size information to the LLC prefetching module. Diamonds indicate decision points. 177

5.8 (A) L2C prefetching module comprised of two generic page size aware (PSA) prefetchers and adaptive logic that dynamically selects between them, (B) selection logic implementation, (C) operation in pseudo-code. Pref-PSA (Pref-PSA-2MB) drives prefetching assuming 4KB (2MB) pages. . . . . . . . . . . . . 178

5.9 Heatmap presenting the total allocated memory mapped to 2MB pages across the considered workloads when running on Champsim and on an Intel Xeon E5-2687W machine. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182

5.10 Performance comparison between the PSA, PSA-2MB, and PSA-SD versions of the SPP prefetcher. The speedups are computed over the original implementation of SPP that considers only 4KB pages and stops prefetching at 4KB physical page boundaries [172]. Higher is better. . . . . . . . . . . . . . . . . . . 186

5.11 Performance comparison between the PSA, PSA-2MB, and PSA-SD versions of state-of-the-art L2C prefetchers, across all considered benchmark suites. The speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 5.10. Higher is better. . . . . . . . . . . 188

5.12 Impact of PSA and PSA-SD versions of SPP on performance, L2C/LLC access latency, L2C/LLC miss coverage, and L2C/LLC prefetching accuracy. All results are computed over the original implementation of SPP. For all considered metrics higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191
5.13 Performance comparison between different implementations of the selection logic of the considered prefetchers' PSA-SD versions. Prefetcher BOP is excluded from this experiment since all BOP versions (PSA, PSA-2MB, PSA-SD) provide the same speedups because BOP does not use any structure indexed with the page size. The reported speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 5.10. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 193
5.14 Impact of L2C MSHR size (A), LLC size (B), and DRAM bandwidth (C) on the performance of the PSA and PSA-SD versions of the considered L2C prefetchers. Results are computed over the original implementations of the considered L2C prefetchers. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . 195
5.15 Comparison with state-of-the-art L1d prefetching. NL refers to a next-line L1d prefetcher. IPCP is the state-of-the-art L1d prefetcher. IPCP++ refers to a version of IPCP that freely crosses 4KB page boundaries for prefetching. Higher is better. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 196
5.16 Distribution depicted with violin plots showing the 4-core speedups of the PSA and PSA-SD versions of the considered L2C prefetchers (SPP, VLDP, PPF, BOP) across 100 4-core mixes. Higher is better. . . . . . . . . . . . . . . . . . . 198
5.17 Distribution depicted with violin plots showing the 8-core speedups of the PSA and PSA-SD versions of the considered L2C prefetchers (SPP, VLDP, PPF, BOP) across 100 8-core mixes. Higher is better. . . . . . . . . . . . . . . . . . . 198

8.1 Research vision illustrated with a cartoon. . . . . . . . . . . . . . . . . . . . . . . 210

# List of Tables


3.1 System simulation parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
3.2 Configuration parameters of the previously proposed TLB prefetchers (SP, DP, ASP), the Agile TLB Prefetcher (ATP), the constituent TLB prefetch engines of ATP (STP, H2P, MASP), the Sampling-Based Free TLB Prefetching (SBFP) module, and the Prefetch Buffer (PB) used for storing the prefetched PTEs. The parameters for ATP and SBFP have been empirically selected after thorough sensitivity analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
3.3 Set of statically selected free distances for the considered TLB prefetchers. . . 88
3.4 Input graphs for the GAP benchmark suite [74]. . . . . . . . . . . . . . . . . . . . 88
3.5 Complete set of workloads used for evaluation. . . . . . . . . . . . . . . . . . . . . 89
3.6 Storage budget of the considered TLB prefetchers presented in Section 3.7.2. The storage budgets of the TLB prefetchers include the storage of a 64-entry PB (4928 bits $\simeq$ 0.60KB). The budget of SBFP does not include a 64-entry PB since it is counted in the budget of ATP; our proposal (ATP+SBFP) uses a shared PB. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

4.1 System simulation parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 139
4.2 Configuration parameters of the previously proposed TLB prefetchers (SP, DP, ASP, MP), Morrigan, and the Prefetch Buffer (PB) used for storing the prefetched PTEs. The parameters for Morrigan have been empirically selected after sensitivity analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142

5.1 System simulation parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
5.2 Complete set of workloads used for evaluation. . . . . . . . . . . . . . . . . . . . . 183

5.3 Geometric mean speedups of the PSA, PSA-2MB, and PSA-SD versions of the considered prefetchers across (i) the non memory-intensive workloads from SPEC CPU 2006 and SPEC CPU 2017 benchmark suites (Workload Set 1), (ii) the 80 memory-intensive workloads of Section 5.5.2 (Workload Set 2), and (ii) the 80 memory-intensive workloads plus the non memory-intensive workloads from SPEC CPU 2006 and SPEC CPU 2017 benchmark suites (Workload Set 3). BOP-PSA-2MB and BOP-PSA-SD are excluded from this comparison because BOP does not use any structure indexed with the page size, thus all its page size aware versions are identical. The speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 5.10. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

# 1 Introduction

## 1.1 Memory Wall

Processor performance has featured exponential increase due to groundbreaking innovations realized in domains like $\mu$architecture, circuits, and fabrication technologies throughout the past five decades [114]. However, this is not the case for main memory speeds, which have experienced only nominal improvements over this period [215]. This disparity between processor and main memory speeds significantly limits systems' performance and it is widely known as the *Memory Wall* [193, 295].

To attenuate the Memory Wall bottleneck, computer architects primarily implemented a hierarchy of cache memories that trade off capacity for speed; caches closer to a core are smaller but faster than the ones closer to main memory [148]. Cache structures accommodate program data and instructions (*i.e.*, memory blocks) in the hope that they will be requested again in the near future. The rationale behind the integration of cache hierarchies into real-world designs is that storing the most recently used memory blocks into small and fast structures would avoid long-latency main memory accesses upon requests

for these memory blocks, giving the illusion to the application that main memory accesses require only a few cycles to complete. The concept of caching relies on two types of memory reference locality [215]; spatial locality and temporal locality. The former refers to the similarity of access patterns across different memory regions while the latter assumes that there will be a replay of memory accesses in the near future.

Although on-chip cache hierarchies bring significant performance enhancements, especially for workloads exhibiting good locality and regular access patterns, they are incapable of eliminating the Memory Wall bottleneck because they are limited in capacity due to the overhead of implementing large SRAMs near cores. Consequently, computer architects employ additional latency-tolerance techniques (*e.g.*, instruction level parallelism, out-of-order execution, and speculative execution among others) to alleviate this bottleneck.

## 1.2 Hardware Prefetching: Endless Opportunity Domain

One latency-tolerance technique that computer architects widely apply to hide the latency cost of memory accesses is *hardware prefetching* for the cache hierarchy. The core idea behind hardware cache prefetching is to proactively fetch data blocks into the cache hierarchy before a core explicitly requests them, alleviating the pressure placed on the memory subsystem by contemporary applications with massive working set sizes that caches cannot fully store [114, 197, 203, 227, 247].

Modern processor designs heavily rely on predictive schemes when making important control and data flow execution decisions to maintain forward progress while awaiting resolving instructions [197, 198, 232, 301]. In practice, fundamental computer architecture concepts like instruction-level parallelism and out-of-order execution heavily depend on the effectiveness of $\mu$architectural prefetchers and predictors. Without predictive schemes, processor architectures might still be relegated to the age of stall-heavy sequential execution. This is the reason why hardware prefetching has been a very popular research domain with myriads of hardware prefetchers being proposed in recent literature [61, 62, 75, 76, 80, 116, 143, 144, 150, 159, 172, 195, 206, 217, 225, 251, 260, 262, 263, 265, 293, 294, 303]; from simple next-line, next-N-line, and stream prefetchers [105, 116, 260] to sophisticated and complex prefetching modules [75, 80, 172, 293, 294, 303] that target different types of memory access patterns.

Hardware prefetching has the potential to provide outstanding performance and energy enhancements but its benefits heavily depend on the properties and the implementation details of the underlying prefetcher. At the extreme, an oracle hardware prefetcher would proactively fetch all useful memory blocks into the first level cache at zero cost, thus all memory accesses would find the requested memory blocks in the first level of the cache hierarchy, avoiding energy and latency-hungry main memory accesses. However, real-world hardware prefetchers are far from approaching the performance of the oracle hardware prefetcher because their prefetch engines cannot identify all possible access patterns in a timely manner, thus some prefetches are either early or late or completely inaccurate, wasting useful resources such as energy, bandwidth, and cache capacity. Finally, hardware prefetching may even harm performance when the underlying prefetcher fails at detecting the memory access patterns of the application but keeps issuing prefetch requests.

Designing a hardware prefetcher capable of identifying the majority of the memory access patterns across different types of workloads is a very ambitious and challenging task because the prefetcher should concurrently address three demanding challenges in order to effectively hide the processor-memory performance gap: (i) *What* to prefetch, (ii) *When* to prefetch, and (iii) *Where* to prefetch. In other words, the hardware prefetcher should (i) be effective and accurate by issuing prefetch requests that mostly provide cache hits–What, (ii) ensure timely fetched prefetched blocks and avoid late prefetches by predicting when to issue prefetch requests–When, and (iii) select in which data structure the prefetched blocks should be stored and, potentially, which blocks should be victimized–Where.

Effective hardware prefetching has proven successful in attenuating the Memory Wall bottleneck; this is the reason why real-world designs employ various hardware prefetchers at different levels of the cache hierarchy [23, 126, 133, 179, 242, 256, 261, 266]. However, the details of the different hardware prefetchers integrated in real-world implementations are not publicly available for most processor vendors. Despite myriads of prior research works in the domain of hardware prefetching [61, 62, 75, 76, 80, 116, 143, 144, 150, 159, 172, 195, 206, 217, 225, 251, 260, 262, 263, 265, 293, 294, 303], there is still potential for large performance enhancements by designing intelligent hardware prefetching modules, composed of one or more prefetch engines, capable of gracefully handling all challenges of hardware prefetching (What to prefetch⟷When to prefetch⟷Where to prefetch) since existing hardware prefetchers' performance is far from approaching the performance of the oracle hardware prefetcher.

## 1.3 Virtual Memory Systems

Virtual memory is a memory management technique that provides an idealized abstraction of the available storage resources on a given machine [87], creating the illusion to programmers of enormous main memory sizes. In virtual memory systems, programs solely operate on virtual addresses, thus, when they issue a memory request, a translation from the virtual address space to the physical address space needs to be performed.

Virtual memory provides unique benefits that are vital for the success of computing; it increases programmer productivity by removing the need to deal with the complexity of the physical address space while ensuring process protection and isolation, among others. Someone could understand the importance of virtual memory for computing by trying to answer the following questions: (i) *Do you think about the existence of virtual memory when writing code today?* (ii) *Would you keep on writing code in the absence of virtual memory?*

Nothing comes for free and virtual memory is not an exception. Programs perform memory accesses using virtual addresses, thus each memory access requires a translation from the virtual to the physical address space. This address translation incurs high performance and energy overheads [55, 58, 66, 71, 82, 84, 99, 117, 166, 175, 202, 237] because it is on the critical path of memory accesses, as the physical addresses must be determined before accessing the actual data. Despite the overheads, processor vendors implement virtual memory due to its unique benefits and they try to reduce the virtual memory overheads by providing both software and hardware support dedicated to address translation.

### 1.3.1 Basic Architectural Support for Address Translation

Modern virtual memory implementations rely on *paging*, a strategy that divides the virtual and the physical address spaces into *pages*, a fixed-size chunk of contiguous memory managed as a single unit. All the bookkeeping of virtual memory metadata is done by the *Page Table*, a per-process data structure that stores the virtual to physical mappings for all pages loaded to main memory. On the hardware side, the *Memory Management Unit (MMU)* accelerates address translation through the *Translation Lookaside Buffer (TLB)*, a hardware structure dedicated for accelerating address translation. However, real-world implementations employ advanced hardware support for address translation; Section 1.3.2 provides an overview of such advanced support.

**Page Table** is an OS structure that accommodates the virtual to physical mappings for each process in the system. Modern page table designs are typically hierarchical; literature refers to hierarchical page tables as *radix tree page tables*. Such designs split the page table into multiple levels. For instance, x86-64 architectures implement 4-level and 5-level radix tree page tables [1, 31]. The building block of the page table is the *Page Table Entry (PTE)*. Each PTE stores all information about a virtual to physical mapping. Upon a memory request, the processor searches the page table to find the corresponding address translation. This procedure is known as *page table walk* or simply *page walk*. Page table walking incurs high latency, bandwidth, and energy overheads and contributes to the Memory Wall bottleneck due to the memory references required for obtaining the requested address translation entries from the page table.

**Translation Lookaside Buffer (TLB)** is a small and private per-core data structure that stores the most recently used PTEs. TLBs accelerate address translation because the processor can rapidly obtain the commonly used address translations without triggering a high-overhead page table walk. Upon a memory access, the processor looks up the TLB for the requested translation. On TLB hits, the address translation is obtained fast. However, on TLB misses, a page table walk is triggered to find the corresponding address translation. Frequent TLB misses significantly harm system performance and enlarge the processor-memory performance gap [55, 71, 82, 84, 237] due to the memory references introduced by traversing the page table to obtain the corresponding address translation entries.

### 1.3.2 Advanced Hardware Support for Address Translation

Processor vendors dedicate additional hardware resources to implement sophisticated modules and policies aimed at reducing the address translation overheads because contemporary applications place tremendous pressure on the virtual memory subsystem due to their massive code and working set sizes. Therefore, contemporary MMU designs consist of multi-level TLB hierarchies with separate first-level TLBs for instructions and data and unified last-level TLBs for both instructions and data, support for multiple page sizes at all TLB levels, hardware page table walkers, and MMU-Caches [142]. Next, we provide a brief overview of these modern MMU components.

**Multi-level TLBs.** The increase in memory footprints of contemporary applications demands for larger TLBs. Simply increasing the size of a monolithic TLB structure would incur higher TLB access latencies, essentially growing the memory access critical path. Therefore, processor vendors employ private per-core multi-level TLB hierarchies; they typically implement two TLB levels per core [5, 9, 10, 31, 142]. L1 TLBs are small with low access latency to ensure fast search for address translations. In addition, there exist separate L1 TLBs for instructions and data. Last-level TLBs are bigger and slower than L1 TLBs and they trade speed for capacity to avoid frequently triggering long-latency page table walks. Finally, last-level TLBs accommodate both instruction and data PTEs into the same structure.

**Multiple Page Sizes.** Modern OSes and architectures increase the effective capacity of the TLB, known as TLB *reach*, by providing support for *superpages* or simply *large pages*, *i.e.*, pages of a size larger than standard 4KB pages. For instance, x86-64 architectures concurrently support 4KB, 2MB, and 1GB pages. Using superpages significantly increases TLB coverage since a single TLB entry for a superpage maps the same memory as multiple TLB entries for standard 4KB pages. However, using page sizes larger than 4KB negatively impacts the flexibility of memory management while providing coarse-grained page protection. Finally, the advent of multiple page size support in modern OSes and architectures asks for non-trivial design choices. Separate or unified L1 TLBs for different page sizes?[1] Separate or unified last-level TLBs for different page sizes? Section 2.3.3 elaborates on the advantages and drawbacks of the different implementation alternatives of supporting multiple page sizes while commenting on the decisions taken by leading processor vendors.

**Hardware Page Table Walkers.** In the absence of hardware page table walkers, the processor was context switching to the OS on every TLB miss [149, 151] that incurs tremendous performance penalties. However, modern architectures perform page walks entirely in hardware, obviating the need for context switching on every TLB miss [67, 71, 86, 120]. Hardware page table walkers are finite state machines that partially hide the latency cost of page table walks by overlapping independent instructions thanks to out-of-order execution while concurrently serving multiple TLB misses [67, 190, 219, 221, 222]; typically up to 4 concurrent TLB misses.

[1] L1 TLBs are already split between instructions and data.

**MMU-Caches.** To reduce the latency cost of page walks, processor vendors employ small and low-latency hardware structures, called *MMU-Caches* [66, 82, 142, 215], that store entries from the intermediate levels of the page table. MMU-Caches bring significant performance and energy enhancements since they enable skipping one or more page table level lookups, essentially reducing the latency cost as well as the memory footprint of page walks. For instance, hitting in all MMU-Caches implies that the corresponding page walk needs to perform only one memory reference for the leaf page table level; without MMU-Caches multiple memory references would be needed to obtain the translation, depending on the page table organization.

## 1.4 Motivation

The performance of page-based virtual memory significantly impacts the overall performance of the system since each memory access requires a translation from the virtual to the physical address space. When address translation is performed efficiently, it enables fast memory accesses and ultimately ensures high system performance. On the other hand, when address translation is performed inefficiently, the memory accesses take many cycles to complete, essentially exacerbating the Memory Wall bottleneck. Therefore, efficient address translation is a prerequisite for efficient and reliable computing.

### 1.4.1 Mismatch in TLB and Memory Sizes

Physical memory is growing exponentially cheaper and larger over the years [69]. This trend has enabled practical physical memory sizes to grow from megabytes to multi-gigabytes and now even to terabytes. This increase in physical memory sizes motivated the advent of emerging workloads with massive data and code footprints that ask for low-latency memory accesses. Therefore, modern systems need to efficiently translate addresses for multi-gigabytes or even terabytes of memory. This ever-growing trend in memory requirements of contemporary workloads stretches the virtual memory subsystem.

Despite the presence of many sophisticated software and hardware schemes dedicated to address translation, some of them were briefly introduced in Section 1.3.2, the virtual memory subsystem still performs poorly when applications with large data and/or code

footprints are executed in the system because the increase in the data/code sizes of these applications outpaces the increase in TLB sizes. The reason for this mismatch is that TLB accesses are on the critical path of accessing memory, necessitating fast access times and low lookup latencies. Therefore, augmenting the TLB structures with additional entries is not a viable and scalable solution; it may improve TLB coverage for some workloads but would also increase the TLB lookup latency as well as the energy consumption of address translation, essentially enlarging the critical path of execution. For this reason, TLB sizes have experienced only nominal growth over the years.

The main takeaway is that TLB sizes are growing at a slower pace than the data and code footprints of contemporary applications, thus, there is a big need for novel designs and implementation approaches capable of effectively hiding this mismatch without incurring high area, storage, and energy overheads.

### 1.4.2 Unexploited Address Translation Metadata

Another key aspect brought by page-based virtual memory systems is the large amount of information and metadata available at the hardware and the runtime system levels, which neither current architectures nor software stacks fully exploit despite their potential for enabling significant performance and energy improvements. Designing efficient and easily implementable modules that propagate the address translation metadata to $\mu$architectural components without direct access to this information has the potential to provide outstanding benefits by (i) enabling $\mu$architectural optimizations that improve performance and energy efficiency, and (ii) opening up new research on designing modules that smartly exploit the address translation metadata available at the $\mu$architecture.

## 1.5 Thesis Approach

Despite groundbreaking technological innovations and revolutions, Memory Wall still limits system performance due to the discrepancy between processor and memory speeds. Virtual memory further aggravates the Memory Wall bottleneck due to the requirement of traversing the memory hierarchy (caches and main memory) multiple times per TLB miss to extract the corresponding translations from the radix tree page table. To make matters worse, the advent of emerging workloads with massive data and code footprints places additional

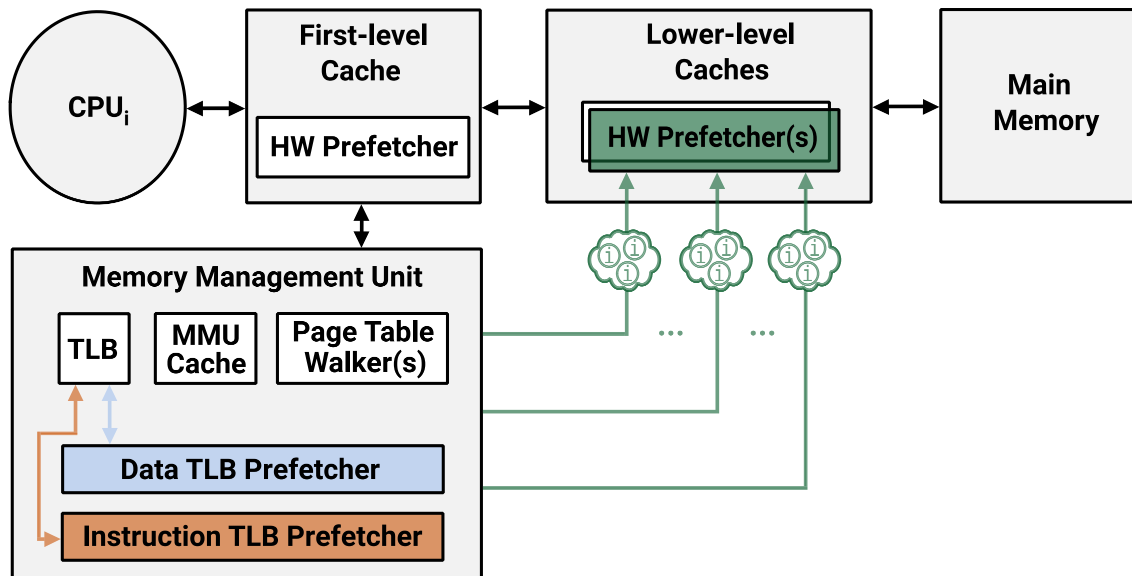


Figure 1.1: Cartoon illustrating the contributions of this thesis integrated into a modern $\mu$architecture. The first contribution (blue) is a composite TLB prefetcher for data accesses. The second contribution (orange) is the first ever TLB prefetcher for instruction accesses. The third contribution (green) is the first work to exploit address translation metadata available at the $\mu$architecture for improving the efficacy of lower-level cache prefetchers.

pressure on the virtual memory subsystem as well as the memory hierarchy, threatening the performance and the efficiency of computing. This thesis aims at reducing these overheads by designing and proposing advanced hardware prefetching mechanisms for page-based virtual memory systems.

We design hardware prefetching modules for the last-level TLBs that consist of multiple low-cost and low-scope prefetchers coupled with additional smart components that dynamically adjust the prefetching strategy. The proposed composite prefetching modules proactively fetch address translation entries into the TLB hierarchy, saving latency and energy-hungry page walks for both data and instruction accesses.

Apart from designing prefetching modules for the last-level TLBs, we demonstrate how to exploit address translation metadata available at the $\mu$architectural levels to improve the efficacy of any hardware cache prefetcher, without modifying the underlying prefetcher's design. Finally, we transparently modify the design of hardware cache prefetchers to inherently use the available address translation metadata and take them into account when driving prefetching decisions.

## 1.6 Thesis Contributions

Motivated by the disparity between TLB sizes and data/code working set sizes of contemporary applications, the first two contributions of this thesis are composite prefetching modules for the last-level TLB, capable of accelerating address translation associated with data and instruction accesses, respectively. The final contribution of this thesis proposes $\mu$architectural techniques that improve the efficacy of cache prefetchers operating in the physical address space by enabling *safe* prefetching beyond standard 4KB physical page boundaries when the system uses large pages, *i.e.*, pages larger than 4KB. Figure 1.1 illustrates the contributions of this thesis integrated into a modern $\mu$architecture. Followingly, we comment on the core idea behind each contribution.

### 1.6.1 Agile Prefetching for the Data TLB Miss Stream

Address translation overheads due to data accesses is a major performance bottleneck in workloads featuring large data working sets [55, 58, 66, 71, 82, 84, 99, 117, 175, 202, 237, 298]. These workloads exacerbate TLB pressure, causing frequent data TLB misses that incur high performance and energy costs due to the page walks required for fetching the corresponding address translation entries.

A variety of prior work seeks to ameliorate the address translation bottleneck for data accesses. These approaches mainly fall into three categories: (i) conservative schemes that increase TLB reach [103, 212, 219, 246, 270, 271], (ii) disruptive approaches [51, 71, 169], and (iii) TLB prefetching mechanisms [85, 165]. Conservative approaches are limited by coalescing opportunities exposed by the application and OS as well as the capacity of the TLB hierarchy. Disruptive approaches call for a radical re-engineering of the virtual memory subsystem, hindering their adoption and possibly introducing new security vulnerabilities. Finally, TLB prefetching is a pure $\mu$architectural technique that relies only on the memory access patterns of the application, is independent of the system state (OS, load, fragmentation), is not disruptive for the virtual memory subsystem, and does not imply any OS involvement. Surprisingly, we found that there has been a scarcity of research in TLB prefetching for over a decade.

In this work, we take a fresh look at $\mu$architectural TLB prefetching for data accesses. First, we analyze and evaluate previously proposed data prefetchers for the second-level

TLB, using an extensive set of academic and industrial workloads. Our analysis extracts the following key findings: (i) no single prior TLB prefetcher performs best across all evaluated workloads, *i.e.*, different prefetchers perform best for different workloads, and (ii) some workloads do not benefit from TLB prefetching since they experience highly irregular patterns. For these workloads, previously proposed TLB prefetchers either naively continue to issue prefetches, burdening the memory subsystem with additional page walk references without providing any benefit, or pessimistically avoid issuing prefetch requests missing opportunities for improving system performance.

We further observe that previously proposed TLB prefetchers always trigger a prefetch page walk[1] to prefetch an address translation entry. In this work, we show that exploiting the locality in the last level of the radix tree page table[2] has the potential to significantly enhance the performance of any TLB prefetcher by reducing the number of memory references it triggers to serve prefetch page walks. However, we find that exploiting page table locality for TLB prefetching in a realistic context requires a smart scheme to identify the most useful PTEs per page walk.

The first contribution of this work is *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that exploits page table locality for TLB prefetching. SBFP predicts through sampling which of the cache-line neighboring PTEs per page walk are more likely to prevent future TLB misses and fetches them into a small TLB buffer.[3] We demonstrate that SBFP (i) provides unique benefits over the naive approach that prefetches all cache-line neighboring PTEs per page walk, (ii) can be combined with any TLB prefetcher to achieve notable performance gains while reducing the memory footprint of page walks and the energy consumption of address translation, and (iii) can operate on page walks triggered to serve both demand accesses and prefetch requests.

To address our first two findings –no single prior TLB prefetcher performs best across all workloads, and some workloads are not benefited by TLB prefetching– we design and propose the *Agile TLB Prefetcher (ATP)*, a composite TLB prefetcher for data accesses, implemented as a decision tree, a supervised learning approach [20]. Unlike prior work that correlates patterns using single features (*e.g.*, strides or PCs or distances), ATP considers several features by combining three low-cost and low-scope TLB prefetchers while adapt-

[1]Prefetch page walks are page walks triggered in the background and speculatively fetch PTEs.
[2]At the end of each page walk, the requested PTE is grouped with 7 neighboring PTEs and they are stored into a cache line [86].
[3]TLB prefetchers typically use a buffer to store the prefetched PTEs to avoid polluting the TLB [153, 165].

ing its prefetching strategy depending on the memory access pattern of the application. To do so, ATP relies on two lightweight mechanisms: (i) adaptive selection logic that dynamically activates the most appropriate TLB prefetcher per TLB miss, and (ii) a throttling scheme that disables TLB prefetching during phases when it is not helpful.

Our evaluation considers 150 academic and industrial workloads, spanning different contemporary benchmark suites. Over the best performing prior TLB prefetcher per benchmark suite, ATP+SBFP improves geometric mean performance by up to 8.7%. Finally, ATP+SBFP significantly reduces the page walk references to the memory hierarchy resulting in up to 75% dynamic energy reduction, at a cost of 2KB of storage.

### 1.6.2 Markov-based Prefetching for the Instruction TLB Miss Stream

Address translation associated with data accesses has been established as a major performance bottleneck in HPC, desktop, and big data applications that have massive data working sets, as highlighted in our first contribution, presented in Section 1.6.1. However, the address translation overheads of instruction accesses have been neglected because these applications feature small code footprints that fit in the TLB hierarchy. In response, the research community has been mainly focused on reducing the address translation overheads of data accesses, omitting instruction address translation.

Recent academic and industrial studies [166, 209, 304] demonstrate that modern server and datacenter applications feature not only large datasets but also large code footprints owing to their huge binaries and deep software stacks. As a result, these applications place tremendous pressure on processor front-end structures (*e.g.*, iCache, iTLB), compromising performance due to unavoidable pipeline stalls. To make matters worse, the front-end performance bottleneck is likely to be aggravated since the instruction footprint of modern server and datacenter applications is annually increasing within the 20-30% range [166], significantly outpacing the growth in the front-end structure sizes.

In this work, we argue that instruction address translation is an emerging performance bottleneck in server applications. To support our claim, we provide the first $\mu$architectural study that (i) characterizes the TLB behavior of industrial server workloads, and (ii) provides evidence that instruction address translation is a performance bottleneck in servers.

Surprisingly, minimal attention has been paid to the instruction address translation costs of server and datacenter applications. Existing software approaches comprise either code

layout optimization techniques [207] or OS schemes leveraging large pages [304]. On the hardware side, there is no prior work specifically targeting the instruction address translation bottleneck. Previously proposed hardware schemes, originally conceived to target second-level data TLB misses, could be also effective for instruction TLB misses. Proposals that increase TLB reach [219, 221] have narrow applicability to the instruction address translation problem since they are limited by coalescing opportunities exposed by the application and the OS, and are also susceptible to security problems. Approaches proposing an overhaul of the virtual memory subsystem [51, 71] require radical changes, which seriously limits their adoption and may also bring up new security vulnerabilities. Finally, TLB prefetching constitutes a fully legacy-preserving $\mu$architectural technique that relies solely on the memory access patterns of the application, is independent of the system state, and does not disrupt the virtual memory subsystem. However, (i) there is no previously proposed instruction TLB prefetcher, and (ii) the effectiveness of prior data TLB prefetchers on prefetching for the instruction TLB miss stream has never been analyzed.

Our analysis on a set of 45 industrial server workloads draws the following key findings: (i) state-of-the-art designs of data TLB prefetchers are mainly unable to cover instruction TLB misses, (ii) instruction TLB misses follow a skewed distribution, with a modest number of pages responsible for the majority of the misses, (iii) instruction TLB miss patterns are mostly irregular while having limited spatial locality restricted to a small region around the triggering miss, and (iv) the instruction TLB miss stream correlates well with the miss frequency of instruction pages.

To address these findings, we propose *Morrigan*, the first $\mu$architectural TLB prefetcher for instruction references. Morrigan consists of two complementary modules to capture both irregular and regular patterns: (i) an ensemble of table-based hardware Markov prefetchers that efficiently build and store variable length Markov chains out of the instruction TLB miss stream while using a new frequency-based replacement policy to manage their predictive structures, and (ii) a sequential prefetcher that operates only when the Markov prefetchers are unable to produce prefetch requests. Finally, both modules of Morrigan exploit the locality in the last level of the radix tree page table [86], similar to our first contribution presented in Section 1.6.1, to further improve miss coverage.

On a set of 45 industrial server workloads, Morrigan eliminates 69% of the memory references in demand page walks triggered by instruction TLB misses and achieves 7.6% geometric mean speedup over a baseline without instruction TLB prefetching.

### 1.6.3 Safe Cache Prefetching Beyond 4KB Page Boundaries

The increase in working set sizes of contemporary applications outpaces the growth in cache sizes, resulting in frequent main memory accesses that deteriorate system performance due to the disparity between processor and memory speeds [193, 295].

Prefetching is an effective technique that hides the latency cost of memory accesses by proactively fetching data blocks into the cache hierarchy before a core explicitly demands them, alleviating the pressure placed on the memory subsystem by applications with large data working sets that the caches cannot fully contain [197].

Previously proposed cache prefetchers generally fall into two categories; spatial prefetchers [61, 75, 143, 144, 172, 195, 251, 262] and temporal prefetchers [62, 150, 176, 293, 294]. The former exploits the similarity of access patterns across different memory regions to drive prefetching. In contrast, the latter do so by recording sequences of past cache misses. Although effective, temporal cache prefetchers have major drawbacks compared to spatial prefetchers: (i) spatial prefetchers require orders of magnitude less metadata storage compared to temporal prefetchers [61], (ii) spatial prefetchers can save compulsory misses [263] whereas temporal prefetchers are fundamentally limited to prefetch for compulsory misses, and (iii) spatial prefetchers not only save long-latency cache misses but also improve the system energy consumption by eliminating DRAM row activations [61, 143, 289].

Our analysis on four state-of-the-art spatial prefetchers operating in the physical address space, using both academic and industrial workloads, drives the following key findings:

- Previously proposed spatial cache prefetchers operating in the physical address space do not permit prefetching beyond 4KB physical page boundaries since physical address contiguity is not guaranteed and crossing 4KB physical page boundaries for prefetching is susceptible to security issues since physical address contiguity is not guaranteed while an adversary could exploit page-crossing prefetching to create a side-channel [128, 287].

- Modern systems predominately use large pages, *i.e.*, pages larger than base 4KB pages, when executing memory-intensive workloads.

- Previously proposed cache prefetchers do not take into account the existence of large pages when driving prefetching decisions.

- Making lower-level cache prefetchers aware of the page size information has the potential to provide great benefits by enabling safe prefetching beyond 4KB physical page boundaries when the accessed blocks reside in large pages.

- Integrating the notion of large pages into the design of a cache prefetcher may positively or negatively impact performance, depending on the properties of the workload.

Based on our findings, we propose the *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that exploits modern prevalence and support for large pages to improve the efficacy of lower-level cache prefetching. PPM propagates the page size information to the lower-level cache prefetchers, enabling safe prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page. In practice, PPM exploits the available address translation metadata after a first-level cache miss and directs the page size information to the lower-level cache prefetchers through the first-level caches' Miss Status Holding Registers (MSHRs). PPM is a fully legacy-preserving scheme that operates without requiring any costly TLB lookup or reverse address translation and can be used to enhance the performance of any lower-level cache prefetcher without requiring design modifications. We refer to a prefetcher that exploits PPM as *Page Size Aware Prefetcher (Pref-PSA)*.[1] Note that a Pref-PSA inherently uses 4KB pages to drive prefetching decisions since PPM enables beyond 4KB physical page boundaries prefetching (when possible) without modifying the prefetcher's design.

We capitalize on the benefits of PPM by transparently integrating the notion of large pages into the design of any cache prefetcher. We observe that doing so may positively or negatively impact performance as some workloads enjoy great benefits by making the prefetcher inherently use large pages while others experience performance degradation because large pages provide a coarser representation of the memory access patterns than standard 4KB pages. To avoid harming performance while exploiting the benefits of integrating large pages in the prefetcher's design, we implement a composite prefetching scheme that consists of two identical versions of the same Pref-PSA. The prefetchers differ in only one aspect; one Pref-PSA inherently uses standard 4KB pages to drive prefetching while the other Pref-PSA uses large pages.[2] In addition, the composite scheme uses adaptive selec-

[1] It can be any cache prefetcher operating in the physical address space.
[2] We focus on 2MB large pages since Linux provides automatic and transparent support, using the Transparent Huge Pages (THP) mechanism [43], only for 2MB large pages.

tion logic based on Set-Dueling [228] to dynamically enable the most prominent of the two competing prefetchers. We refer to this composite prefetching scheme as *Pref-PSA-SD*.

To highlight our proposals' versatility, we apply the proposed page size exploitation techniques at four state-of-the-art L2C prefetchers [80, 172, 195, 251]. Our evaluation shows that our proposals (i) improve geometric mean performance by up to 8.1% over the original versions of the considered prefetchers, across 80 memory-intensive workloads and different core configurations, by significantly improving the timeliness and the coverage of the prefetchers and (ii) do not harm the performance of non-memory-intensive workloads.

# 2 Background

This chapter provides background on the Memory Wall bottleneck while presenting the fundamental idea, the properties, and the challenges of hardware cache prefetching.[1] Next, it presents the basics of virtual memory, discussing different implementation strategies. Followingly, it compares conventional and modern architectural support for page-based virtual memory systems while highlighting the impact of virtual memory on the Memory Wall bottleneck. Although architectural support for address translation is similar among different architectures (*e.g.*, x86 [5, 30], ARM [9], RISC-V [38]), this thesis mainly focuses on x86-64 architectural support for address translation. Therefore, this chapter elaborates on how x86-64 address translation is performed. Finally, it presents the basics of hardware prefetching for the $\mu$architectural structures of the virtual memory subsystem while commenting on the impact of virtual memory on hardware cache prefetching. Note that the problems tackled by this thesis are not architecture specific, thus our contributions can be implemented in any other contemporary architecture with virtual memory support.

[1] This thesis focuses on 3-level cache hierarchies. The contributions of this thesis are not dependent on the number of cache levels.

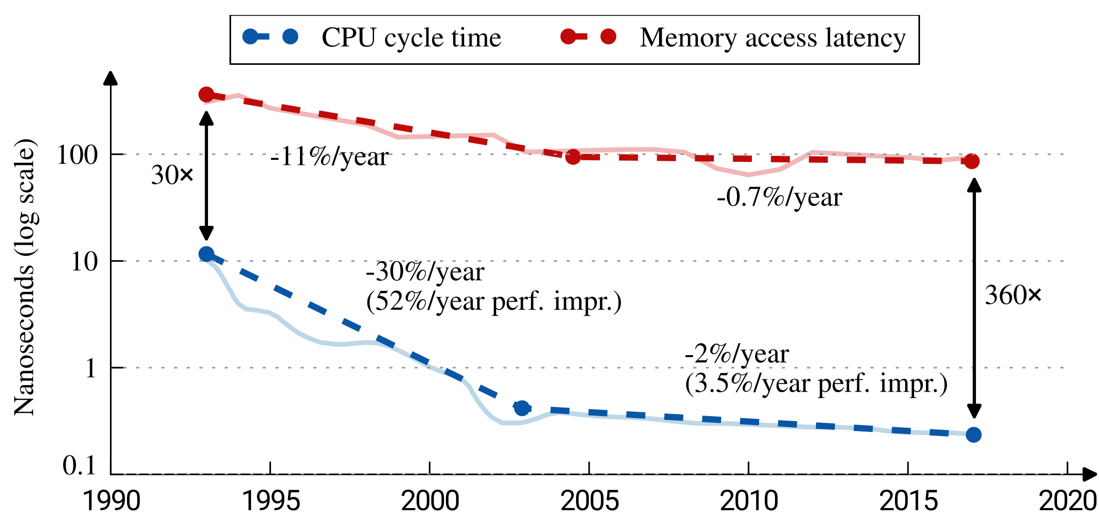


Figure 2.1: Disparity between processor and main memory speeds for the last 25 years.

## 2.1 Memory Wall

Throughout the last five decades, there have been many significant innovations in domains of computer architecture [114]; from the $\mu$architecture to circuits and fabrication technologies. These innovations have led to significant improvements in processors' performance. Although processor speeds have experienced exponential increases, this is not the case for main memory speeds which have featured only nominal increases [215]. Back in 1995 Wulf and McKee published an article entitled *'Hitting the Memory Wall: Implications of the Obvious'* [295] which (i) annotates that the computer architecture community was mainly focused on improving processor performance, ignoring the existence of the main memory systems, and (ii) projects the processor-memory performance gap to highlight that this gap would keep on increasing, making main memory accesses a major bottleneck for the next-generation computing systems. The title of this article [295] is the reason why the discrepancy between processor and main memory speeds is widely known as the *Memory Wall*. Figure 2.1 depicts the Memory Wall bottleneck by showing the disparity between main memory access latencies and CPU cycle times for the last 25 years [229]. It can be observed that, in recent years, single-core performance experienced 3.5% annual increase [215] while memory access latency decreases by 0.7% per year. The main takeaway is that even if the processor-memory gap stops increasing at the same rate, the current performance gap is large enough to necessitate mitigation techniques.

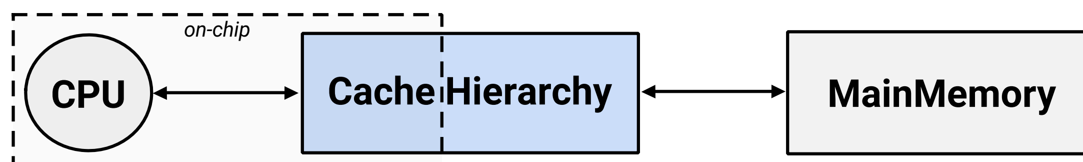


Figure 2.2: Hierarchy of cache memories. Caches can be placed either on-chip or off-chip.

The computer architecture community has employed various latency-tolerance techniques to alleviate the Memory Wall bottleneck over the years. Historically, hierarchies of cache memories is the first attempt toward shrinking the processor-memory performance gap. Figure 2.2 illustrates the placement of cache memories in a basic $\mu$architecture. Caches closer to the core are smaller but faster than the ones closer to the main memory [148]. Moreover, caches can be placed either inside or outside the chip. The decisions regarding the design, the size, the placement, and the properties of the different cache memories are taken by the responsible computer architect(s).

The concept of caching relies on two types of memory reference locality [215]: *spatial locality* and *temporal locality*. Spatial locality refers to the similarity of access patterns across different memory regions while temporal locality assumes that recently utilized memory blocks will be utilized again within a relatively small time window. In practice, cache memories provide temporary storage for program data and instructions (*i.e.*, memory blocks), hoping that these blocks will be requested by forthcoming memory accesses. The rationale behind the concept of caching is simple and straightforward: accommodating recently used memory blocks into small and fast structures would avoid long-latency main memory accesses upon requests for these memory blocks, giving the illusion that main memory accesses are served within a few cpu cycles.

Processor vendors typically employ three-level cache hierarchies in contemporary $\mu$architectural designs. First-level and second-level caches are referred to as L1 cache (L1C) and L2 cache (L2C), respectively, while the third-level cache is called last-level cache (LLC). L1 caches are split between data (L1d) and instructions (L1i) since data and instructions accesses exhibit different locality and reuse attributes. On the other hand, L2Cs and LLCs are typically unified among data and instructions. Moreover, L1 and L2 caches are relatively small and private per core while LLCs are big and shared among the cores of the system. Figure 2.3 shows the anatomy of a modern cache hierarchy, including latency and capacity specifications for a recent Intel Icelake chip [29].

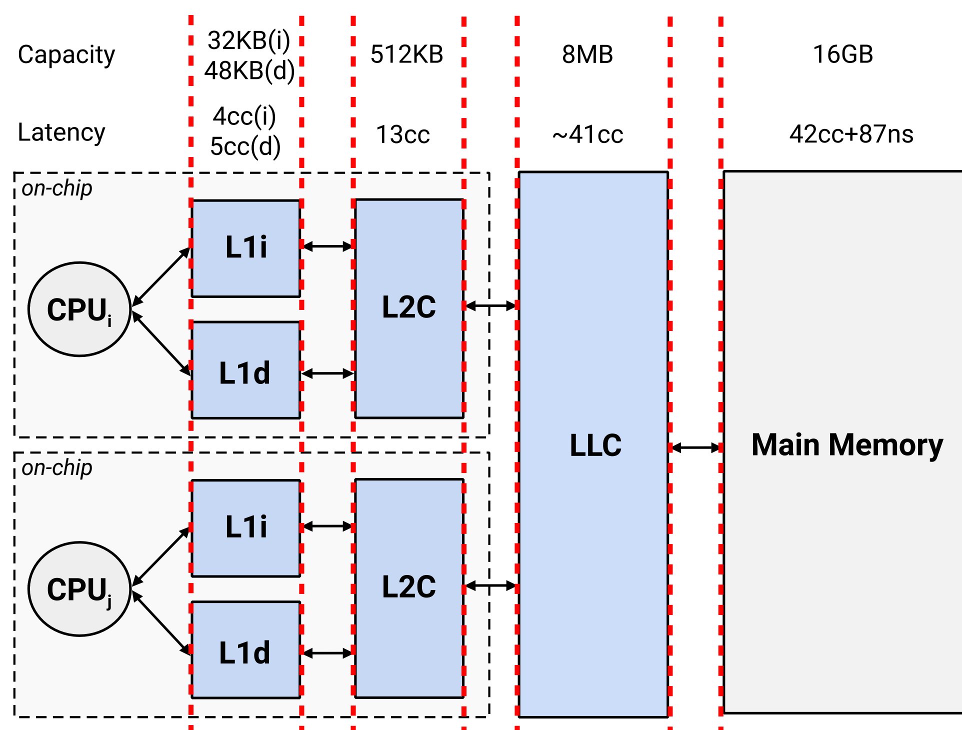


Figure 2.3: Anatomy of a modern cache hierarchy composed of three cache levels. The latency and capacity parameters are extracted from a recent Intel Icelake chip [29].

Modern cache hierarchies partially reduce the processor-memory performance gap, especially for workloads featuring good memory reference locality, providing remarkable performance benefits. However, they fall short at eliminating the Memory Wall bottleneck since they rely on two premises that do not hold for all workload types and memory access patterns. First, they are limited in capacity due to the overhead and the cost of implementing large SRAMs near cores. Second, there is no single optimal strategy for storing and replacing entries in a cache memory that is suitable for all workloads since different workloads exhibit various memory access patterns and locality attributes. The main takeaway is that cache hierarchies are just the first step toward 'knocking down' the Memory Wall.

To further bridge the processor-memory gap, computer architects back up modern multi-level cache hierarchies with cross-layer latency-tolerance techniques such as instruction level parallelism, out-of-order execution, speculative execution, and code and data layout optimizations, among others. The next section presents on one of the most widely deployed latency-tolerance techniques, hardware prefetching.

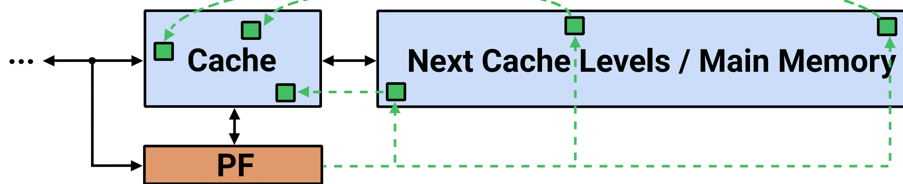


Figure 2.4: Operation of a generic hardware prefetcher (PF) placed alongside a cache memory.

## 2.2 Hardware Prefetching

Hardware prefetching is a speculation technique aimed at tolerating memory access latency and is widely applied due to its potential for reducing the Memory Wall bottleneck. The rationale behind hardware prefetching is simple: proactively fetch data blocks closer to a core before they are explicitly demanded by the application to avoid long-latency main memory accesses. Hardware prefetching is an appealing research domain due to its unique properties; it is a pure $\mu$architectural technique that relies only on the memory access patterns of the application, is independent of the system state, does not disrupt the existing cache subsystem, and does not require any OS involvement.

Figure 2.4 depicts the operation of a generic hardware prefetcher placed alongside a cache memory. Upon a cache access, the prefetcher takes as input the currently requested memory block and issues prefetch requests that speculatively fetch other memory blocks either from the next levels of the cache hierarchy or the main memory into the cache where the prefetcher is placed. Finally, hardware prefetching can be applied at any level of a modern cache hierarchy, as Figure 2.5 shows. Indeed, this is the most common case in modern $\mu$architectural designs today [23, 126, 133, 179, 242, 256, 261, 266] since processors aim at capturing heterogeneous memory access patterns across different types of workloads; Section 2.2.5 discusses hardware prefetching applied in real-world chips.

A prefetch might be accurate or inaccurate. The former reduces the processor-memory performance gap by (partially) hiding latency and energy-hungry main memory accesses. However, an accurate prefetch might be *late*, *early*, or *on-time*. Late prefetches defeat the purpose of pre-fetching since the memory blocks are not fetched by the time a core requests them. Early prefetches are the ones fetching memory blocks that will be demanded at some point in the future, but not from the next memory access, resulting in suboptimal cache management. Finally, on-time prefetches are the ones that fetch memory blocks that will

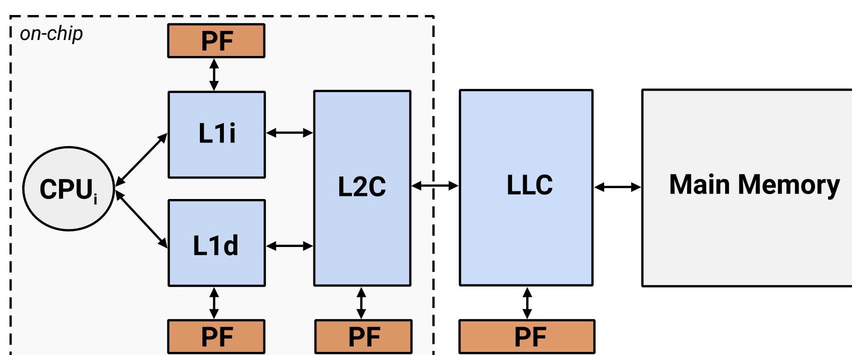


Figure 2.5: Prefetching can be applied at any level of a modern cache hierarchy. PF refers to a generic hardware cache prefetcher.

be requested by the immediately next memory request, entirely hiding the memory access latency while obviating any cache management implication.

### 2.2.1 Oracle Hardware Prefetcher

The benefits offered by a hardware prefetcher heavily depend on its implementation properties. The ideal case would be the oracle hardware prefetcher which would proactively fetch all useful memory blocks into the L1 caches at zero overhead, making all memory accesses hit in the L1 caches. To demonstrate that intelligent hardware prefetching has potential for providing outstanding performance gains, Figure 2.6 shows a simplistic execution diagram comparing the operation of three different prefetching scenarios: (i) no hardware prefetcher, (ii) oracle hardware prefetcher, and (iii) real-world hardware prefetcher. Figure 2.6 breaks down the execution into computation, cache lookup, and main memory access while annotating which prefetches are late, early, on-time, and inaccurate.

In the absence of a hardware prefetcher (Figure 2.6 (A)) we observe that when a cache miss occurs, the processor waits until the memory access is completed to continue computation. On the other hand, the oracle hardware prefetcher (Figure 2.6 (B)) always issues on-time prefetches, *i.e.*, prefetches that timely store in the cache the memory block that will be requested by the next memory request, significantly reducing the execution time. Finally, the real-world cache prefetcher (Figure 2.6 (C)) is far from approaching the performance of the oracle prefetcher because it is incapable of identifying all possible access patterns in a timely manner, thus some of its prefetchers are late, early, or even completely inaccurate while on few of them are on-time.

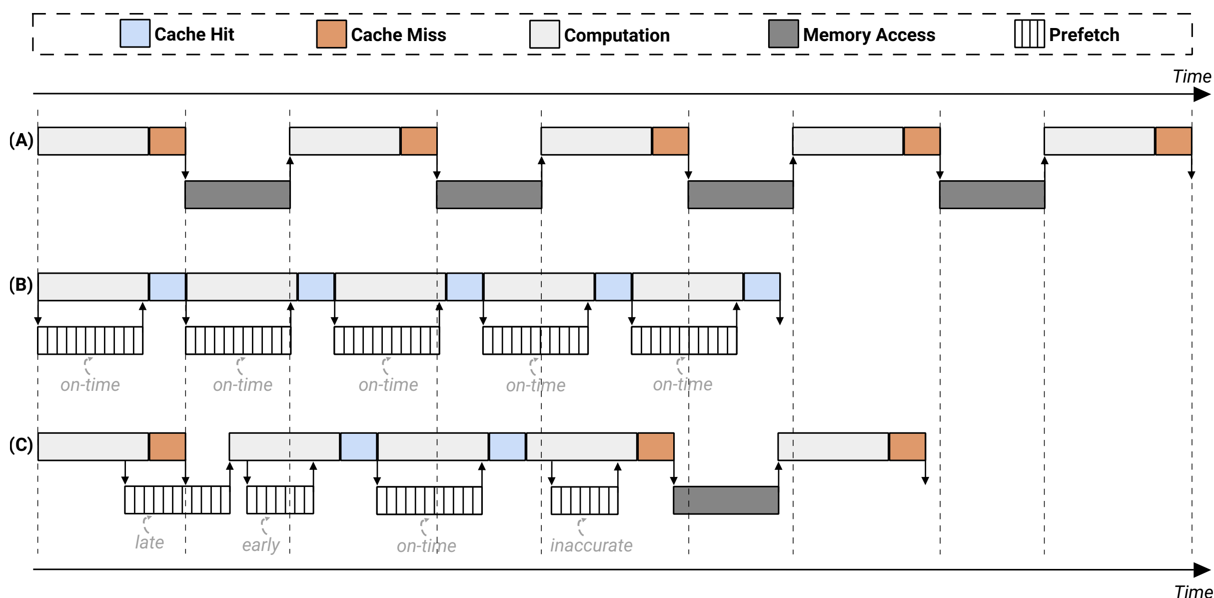


Figure 2.6: Simplistic execution diagram of a system with one-level cache hierarchy equipped (A) without any hardware prefetcher, (B) with the oracle hardware prefetcher, and (C) with a real-world hardware prefetcher. Prefetch requests are classified into on-time, late, early, and inaccurate.

### 2.2.2 W$^3$ Challenge

To hide memory access latency effectively, hardware cache prefetching modules need to be capable of identifying and prefetching memory access patterns across a broad spectrum of workload types. Designing such intelligent prefetching modules is a very ambitious task due to the requirement of gracefully handling the *W$^3$ challenge*: (i) *What* to prefetch, (ii) *When* to prefetch, and (iii) *Where* to prefetch.

**What to Prefetch.** A key challenge for a hardware prefetcher is to predict which memory blocks will be requested in the near future, *i.e.*, by subsequent memory accesses. The prefetching algorithm is responsible for dealing with this challenge since it is the component that drives the prefetching decisions, *i.e.*, it determines which prefetch requests will be issued. If the prefetcher predicts addresses correctly, then it will prefetch memory blocks that will eventually save long-latency memory accesses. However, if the prefetcher fails at predicting which blocks will be subsequently accessed, then the prefetch requests would store memory blocks in the cache that would not serve any memory request. In such cases, the prefetcher wastes resources (*e.g.*, bandwidth, energy) due to fetching useless memory

blocks, generates excessive traffic and contention in the memory system and pollutes the cache structure by evicting potentially useful blocks to store the prefetched blocks.

**When to Prefetch.** Ideally, a prefetcher should precisely predict when to issue a prefetch *i.e.*, predict how many cycles in advance a prefetch should be issued so as to hide all memory access latency. In this best case, all prefetches will be on-time. However, this is not always the case since prefetches might be late or early, as explained in Section 2.2.1. Early prefetching has two major drawbacks; it clutters storage since the prefetched memory blocks might be evicted from the cache before they are requested from the core and may evict useful blocks from the cache that on-time prefetches would not, increasing the number of cache misses observed. On the other hand, late prefetches diminish the effectiveness and defeat the purpose of prefetching since they fail at hiding all memory access latency and may also lead to performance degradation due to slowing down time-critical demand accesses.

**Where to Prefetch.** Hardware prefetchers place prefetched blocks either directly into the corresponding cache structure or into a small buffer that accommodates only prefetched blocks. Using such a buffer avoids any cache pollution introduced by inaccurate prefetches at the cost of incurring additional storage and area overheads while complicating the cache coherence protocol. Modern cache prefetchers typically select to place prefetched blocks directly into the caches. However, this is not always the case for hardware prefetchers belonging to architectural support for virtual memory, as we discuss in Section 2.4.

### 2.2.3 Efficacy Metrics

Could a hardware prefetcher capable of prefetching for most memory access patterns approach the performance gains of the oracle hardware prefetcher? The answer is no; solely capturing the access patterns is not enough to approach the performance of the oracle prefetcher. A prefetcher that manages to identify the memory access patterns but fails at issuing prefetches in a timely manner without making sure that the prefetched blocks would not evict other useful memory blocks cannot provide significant performance enhancements. The effectiveness of a modern hardware cache prefetcher should be thoroughly examined by taking into account many well-established metrics. The most commonly used metrics are listed below.

**Speedup** quantifies how faster a system would be by using a novel module. The following formula quantifies the speedup offered by a hardware prefetcher.

$$Speedup = \frac{IPC_{\text{without prefetcher}}}{IPC_{\text{with prefetcher}}}$$

**Coverage** quantifies how many of the total cache misses have been eliminated thanks to hardware prefetching. The following formula calculates the coverage for a given prefetcher.

$$Coverage = \frac{used\ prefetches}{total\ cache\ misses}$$

**Accuracy** is a metric that answers the following question: How many prefetches were actually useful? The following formula calculates the accuracy of a cache prefetcher.

$$Accuracy = \frac{used\ prefetches}{total\ prefetches}$$

**Timeliness** refers to the portion of useful prefetches that have been already stored in the cache by the time there was a demand request for them.[1] The following formula estimates the timeliness of a cache prefetcher.

$$Timeliness = \frac{on-time\ prefetches}{used\ prefetches}$$

**Cache Pollution** quantifies whether or not a cache prefetcher evicts useful cache blocks due to prefetch placement. This is a hard metric to quantify and its formula is given below.

$$Cache\ Pollution = \frac{demand\ misses_{\text{with prefetcher}}}{demand\ misses_{\text{without prefetcher}}}$$

**Bandwidth Overhead** answers whether or not a given prefetcher increases the bandwidth consumption. This is not a first-class metric when designing a hardware prefetcher because

[1] This definition of timeliness treats all late (and early) prefetches equally. However, prefetches that are a few cycles late (or early) have a different performance impact than prefetches that are many cycles late (or early).

it may falsely annotate that a prefetcher harms bandwidth consumption when the prefetcher actually utilizes idle bus bandwidth. The following formula calculates bandwidth overhead.

$$Bandwidth\ Overhead = \frac{bandwidth\ consumption_{\text{with prefetcher}}}{bandwidth\ consumption_{\text{without prefetcher}}}$$

Apart from the metrics presented above, there are additional aspects that an architect should consider when designing a novel hardware prefetcher. First, the storage overhead of the prefetcher; typically, it should not exceed a few percent of the corresponding cache memory size. Second, the area overheads, *i.e.*, how much die area it occupies. Third, the implementation complexity, *i.e.*, the logic required for implementing the prefetcher (*e.g.*, adders, multipliers/dividers, comparators). Fourth, the computational complexity, *i.e.*, the amount of computation a prefetcher needs to perform in order to produce prefetch requests. Finally, it is also of high importance to be taken into account the impact of prefetcher on the static and dynamic energy consumption of the system.

Ideally, a hardware prefetcher should adequately satisfy all the above metrics. However, prior cache prefetching works are mainly focused on a subset of these metrics, compromising their impact on the other metrics. For example, cache prefetchers proposed before 2010 were mainly focused on achieving high coverage without caring so much for their accuracy and energy footprint [277]. Nowadays, there is a shift in cache prefetching design trends; most works focus on designing sophisticated prefetching modules, often composed of multiple individual prefetch engines, that aim for both high accuracy and high coverage without exacerbating the area, storage, and energy costs [75, 80, 172, 195].

### 2.2.4 Classification of Prior Work

Intelligent hardware prefetchers have the potential to significantly reduce the Memory Wall bottleneck because they rely only on the memory access patterns of the applications without being affected by the system state. There are myriads of prior works on cache prefetching [61, 62, 75, 76, 80, 116, 143, 144, 150, 159, 172, 195, 206, 217, 225, 251, 260, 262, 263, 265, 293, 294, 303]. This chapter classifies them in two broad categories, depending on the characteristics of their prefetching algorithms, and reveal the fundamental properties of each cluster. We refer interested readers to [114] for a more comprehensive analysis and classification of prior cache prefetching works.

#### 2.2.4.1 Spatial Prefetching

Spatial prefetchers [61, 75, 116, 143, 144, 172, 195, 251, 260, 262] rely on spatial address correlation, *i.e.*, the similarity of access patterns across data structures dispersed throughout memory with similar data layouts. For instance, spatial correlation implies that after accessing memory blocks X,Y of a specific memory region managed as a single unit, then it is likely to visit the same locations X,Y of another memory region managed as a unit.[1]

Spatial locality is a very powerful concept; this is the reason why spatial cache prefetchers have been deployed in real-world chips [54, 92, 126, 179, 242, 256, 275]. The intrinsic properties of spatial prefetchers make them suitable for real-world implementations. Specifically, they learn frequent access patterns by keeping track of deltas/offsets within spatial regions and apply these deltas/offsets to other memory regions with similar behaviors. As a result, spatial prefetchers incur low storage overheads (typically less than a few KBs) since they do not need to track and store individual access streams in their internal structures. Furthermore, the ability to use observed deltas on already seen memory regions to prefetch for unobserved memory regions permits spatial prefetchers to save compulsory misses, which constitute a critical bottleneck in scan-dominated applications [263]. Finally, spatial prefetches that travel all the way to DRAM to find the prefetched blocks typically enjoy row buffer hits, downplaying the fetch order impact while reducing the energy consumption of the overall system [143, 289].

#### 2.2.4.2 Temporal Prefetching

Temporal prefetchers [62, 150, 176, 293, 294] drive prefetching decisions by recording the sequence of past cache misses, assuming that there will be a replay of those misses in the future. Temporal address correlation refers to memory accesses that tend to repeat over time in the same order and it stems from specific program characteristics such as data structures traversals (*e.g.*, linked list) that are stable over time. For example, if at some point in time, an access to block A is followed by an access to block B, then it is likely that we observe the same order of accesses (A→B) in the future.

Temporal locality is another powerful tool in the hands of computer architects and has been leveraged to design hardware prefetchers for real-world chips; an example is IBM Blue Gene/Q [133] which includes a temporal prefetching scheme called list prefetching.

[1] Memory regions managed as single units are called *pages* in virtual memory systems, as Section 2.3 explains.

Temporal prefetchers are ideal for saving chains of cache misses that each of them depends on the data of the previous miss; this type of cache misses is typically encountered in applications accessing memory in a pointer-chasing fashion. This type of cache misses incurs significant performance and energy overheads since they need to be resolved serially [134] and cannot be eliminated by spatial prefetchers since they do not exhibit any spatial correlation. Although effective for a broad spectrum of workload types, temporal prefetchers (i) are fundamentally limited at covering compulsory misses since they drive prefetching decisions based on the recording of previously observed accesses, (ii) access DRAM in a fashion that nominally increases the row buffer hit ratio, and (iii) result in high metadata storage overheads due to the need for recording a large number of data accesses over time. Recent works mitigate the high storage overhead of temporal prefetchers through efficient metadata structure designs [176], back-storing or caching metadata [150] and prefetching it as needed [294], and intelligent metadata replacement policies [293].

### 2.2.5 Cache Prefetchers in Modern Chips

Hardware prefetching has proven successful at reducing the processor-memory performance gap. This is the reason why nearly all modern chips employ not only one but multiple hardware prefetchers at the different cache levels [23, 126, 133, 179, 242, 256, 261, 266]. Although vendors disclose that they use multiple hardware prefetchers to hide the latency cost of frequent memory accesses, they only sometimes share information about the type of prefetchers but they never share the implementation details and the intrinsic properties of their prefetching modules; the only way to extract information is reverse engineering [59].

Intel's Nehalem $\mu$architecture is an example where certain information about the hardware prefetchers are publicly available [179]. Specifically, the Nehalem $\mu$architecture contains four distinct hardware prefetchers to capture heterogeneous access patterns; two prefetchers placed alongside L1d and two prefetchers operating at the L2C. The first L1d prefetcher targets sequential patterns that correlate well with the program counter (PC) whereas the second is a stream prefetcher that aims for long sequences of strided accesses. The L2C prefetching module consists of a stream prefetcher and a spatial prefetcher. The former works similarly to the L1d stream prefetcher while the latter fetches two lines instead of one when the L2C streaming prefetcher is enabled. In a similar spirit, AMD's Cortex-A55 [15] and Cortex-A72 [16] processors employ streaming prefetchers at L1d and L2C.

AMD's EPYC 7003 $\mu$architecture [6] is another example where certain information about the cache prefetchers are publicly available. This processor family employs three L1d prefetchers (L1d stream, L1d stride, L1d region) and two L2C prefetchers (L2C stream, L2C up/down). The L1d stream prefetcher fetches sequential lines (in ascending or descending order) based on the history of memory access patterns. The L1d stride prefetcher leverages the access history of distinct instructions to prefetch cache lines when each access is a constant distance from the previous. The L1d region prefetcher is enabled when the data access for a given instruction tends to be followed by a consistent pattern of other accesses within a localized region. Finally, the L2C stream prefetcher operates similarly to the L1d stream prefetcher while the L2C up/down prefetcher exploits memory access history to determine whether to fetch the next or previous line for all memory accesses.

Samsung Exynos cores are high-performance cores present in Galaxy S smartphones and are implemented in a variety of 14nm and 7nm nodes. A recent article elaborates on selected $\mu$architectural aspects of the Samsung Exynos cores [126]. At the L1d, Exynos cores employ two hardware prefetchers coupled with logic that avoids duplicate prefetches. The first L1d prefetcher is a smart multi-stride prefetcher enhanced with a confidence scheme to scale the degree of prefetching that aims at capturing regular memory access patterns. The second L1d prefetcher is based on a previously proposed academic prefetcher [262] that tracks primary loads to memory regions and records the offsets within these regions to drive prefetching. Finally, Exynos cores have two additional hardware prefetchers operating at the L2C. The first L2C prefetcher is a simple buddy prefetcher that generates a prefetch for its neighbor (buddy) sector on every demand L2C miss. The second L2C prefetcher is a sophisticated prefetcher based on [172] that operates in the physical address space and is capable of properly handling long and complex streams within large structures while not permitting page-crossing prefetching to avoid security issues.

Finally, a recent reverse engineering work [287] studies the security implications of hardware prefetching on recent Apple processors. This work reveals that Apple M1, M1 Max, M1 Pro, and A14 processors employ an irregular cache prefetcher, named Array-of-Pointers (AoP) prefetcher, capable of prefetching for pointer-chasing and indirect access patterns [100, 101, 105, 111, 238, 239, 240, 303]. This work demonstrates that Apple's AoP is different from conventional pointer-chasing prefetchers since it prefetches access patterns such as *A[i] or *A[strd*i], where A is an array indexed by i and strd is a stride. We refer interested readers to the original paper [287].

Figure 2.7: The fundamental idea behind virtual memory systems → each memory access requires a translation from the virtual address space to the physical address space. Dotted lines indicate actions.

## 2.3 Virtual Memory

In a nutshell, virtual memory is a memory management technique that provides an idealized abstraction of the available storage resources on a given machine [87]. Most modern computing systems, from desktops to servers and datacenters, implement *virtual memory* to overcome the problem of limited physical memory and provide an easy-to-use model that ensures high programmers' productivity.

Virtual memory provides unique benefits (programmability, security) that are vital for the success of computing, making its presence ubiquitous in almost all computing systems. Specifically, it allows the programmers to think that their data structures are mapped to a big, flat, and linear virtual address space, avoiding any involvement with the complexity of the physical address space. In addition, eschewing any interaction with the implementation of the physical address space increases software portability since the programmer does not need to write code depending on the physical resources of the system (*e.g.*, RAM capacity). Furthermore, virtual memory provides process isolation while improving process protection by preventing other programs from accessing the memory space of other running programs in the system. Another important aspect is that virtual memory improves the OS efficiency on managing multiple processes by permitting programs to undersubscribe or oversubscribe memory. Finally, virtual memory eases memory migration from one physical address to another (*e.g.*, from DRAM to non-volatile memory or from one socket to another).

Nothing comes for free, and so do the numerous benefits of virtual memory. In virtual memory systems, programs operate on virtual addresses, thus upon memory requests, the processor needs to collaborate with the OS and translate those virtual addresses into physical ones. Figure 2.7 depicts the abstract operation taking place upon memory accesses in virtual memory systems. The requirement of translating virtual addresses into physical ones incurs high overheads in terms of latency and energy consumption because the virtual-to-physical translation is performed on the critical path of accessing memory

[55, 58, 63, 64, 66, 71, 82, 84, 84, 95, 97, 98, 99, 109, 115, 117, 160, 161, 162, 163, 164, 166, 175, 178, 185, 191, 199, 202, 213, 236, 237, 257, 264, 298]. Despite the non-trivial overheads and design implications, almost all real-world systems today implement virtual memory due to its unique benefits while computer architects opt to reduce its overheads by inventing and employing hardware structures (*e.g.*, MMU-Caches [142]) and OS policies (*e.g.*, Address Space Layout Randomization or ASLR [123, 171, 249]) dedicated to address translation. Indeed, there are myriads of prior research works [49, 53, 57, 58, 66, 99, 117, 121, 137, 139, 166, 174, 175, 186, 189, 202, 210, 211, 234, 258, 272] and accredited doctoral dissertations [69, 81, 102, 155, 168, 187, 220] aimed at improving and enhancing architectural support for address translation in recent years.

### 2.3.1 Virtual Memory Implementation

A very important decision that needs to be made by the computer architects who are responsible for designing the virtual memory subsystem is the granularity of the address translation bookkeeping. Fine-grained approaches [103, 177, 212] provide more flexibility when managing memory than coarse-grained approaches [204, 222] at the cost of introducing more storage overhead than the coarse-grained schemes due to the amount of address translation metadata that need to be maintained.

The most popular strategies for implementing virtual memory subsystems are *paging* and *segmentation*. The former divides the virtual and the physical address spaces into *pages*, a fixed-size chunk of contiguous memory managed as a single unit. The latter splits the virtual address space into several big logical segments, typically code, data, and heap [184]. Segmentation has one advantage and two major drawbacks compared to paging. The advantage is that segmentation offers space efficiency due to the large size of the segments, making very small and fast TLBs sufficient even for applications with massive memory footprints. However, the size of the segments does not permit fine-grained memory protection [87, 127, 222, 291] while sometimes leading to memory fragmentation [77, 78, 87, 204, 299], *i.e.*, when inefficient mapping of memory regions lead to wasted/inaccessible memory regions.

The majority of modern virtual memory implementations rely on paging due to the inefficiencies of segmentation. For this reason, this thesis targets to improve the performance of page-based virtual memory systems.

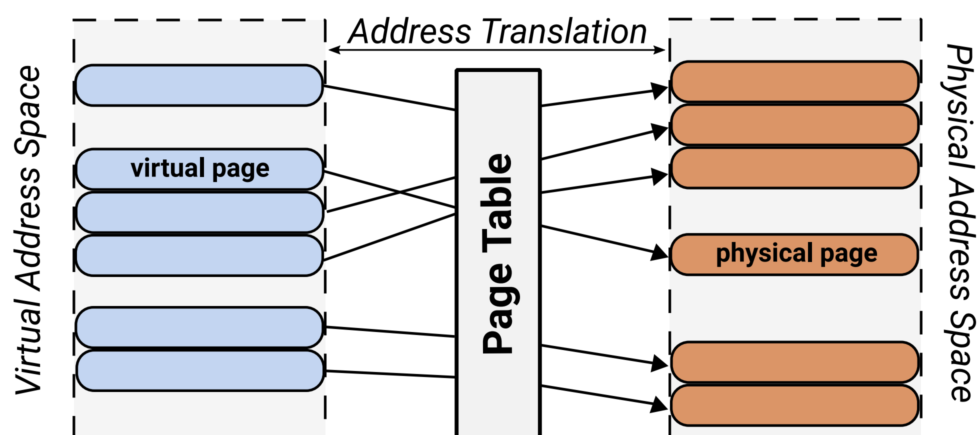


Figure 2.8: Cartoon depicting the abstraction layer of page-based virtual memory. Blue and orange boxes represent virtual and physical pages, respectively. Arrows representing virtual to physical mappings are one-to-one for visibility; this is not necessarily the case due to *homonyms* and *synonyms* [87]. Homonyms refer to cases where on virtual address points to multiple physical addresses. Synonyms refer to cases where multiple virtual addresses point to a single physical address.

### 2.3.2 Building Blocks of Page-based Virtual Memory

This section describes the fundamental concepts that form the base for both conventional and modern page-based virtual memory systems, as shown in Figure 2.8.

**Virtual address space** is the set of memory addresses visible to the process. Programs perform memory accesses using virtual addresses.

**Physical address space** is the set of memory addresses that specify where the actual data are stored. The physical address space is not the physical memory since the former refers to ranges of memory addresses whereas the latter refers to actual storage. Upon a memory access, the OS is responsible to translate the virtual address of the request to a physical address and make available the corresponding virtual to physical mapping.

**Canonical form** refers to the requirement of forcing specific bits of the virtual address to be the same. For example, x86-64 and ARM architectures require bits 48-63 to be the same;[1] accessing a virtual address that is not in canonical form would kick in an exception.

[1] x86-64 architectures are moving towards 57-bit virtual address spaces [1]. In this case, the canonical form would imply bits 57-63 to be the same.

**Paging** is a strategy that divides the virtual and the physical address spaces into fixed-size chunks of contiguous memory managed as individual units, named *pages*. Modern implementations use some form of *demand paging*, a strategy that brings pages into the main memory when they are explicitly requested by the program.

**Page** is a chunk of memory managed as a single unit. Both virtual and physical address spaces are split into pages. Virtual and physical pages are identified by the virtual and physical page numbers, respectively. The virtual (physical) page numbers can be obtained by logically shifting the virtual (physical) address by a number that depends on the actual page size; base pages are typically 4KB but this varies between different processor vendors. A single application may use multiple pages but there is no guarantee that these pages would be contiguously allocated.

**Page Table** is a private per process OS-managed data structure that stores all the virtual to physical mappings for all pages loaded to main memory coupled with the necessary address translation metadata.

**Page Table Entry (PTE)** is the building block of the page table. Each PTE accommodates all required information and metadata for a virtual to physical mapping in the most compact way to avoid inflaming the memory requirements of the page table.

**Page Table Walk** is the procedure that searches the page table to find the translation of a virtual page. Page table walks (generally known as page walks) can be performed either in software or hardware.

**Address translation** is the process followed upon a memory request; the processor collaborates with the OS to translate the virtual address into a physical address. In practice, during address translation, the processor looks up the page table to find the corresponding translation. The process of traversing the page table to find the requested address translation is known as *page table walk* or simply *page walk*. Depending on the implementation, page table walking can be performed either on software or hardware. Section 2.3.3.1 describes both options as well as what is currently implemented in modern designs.

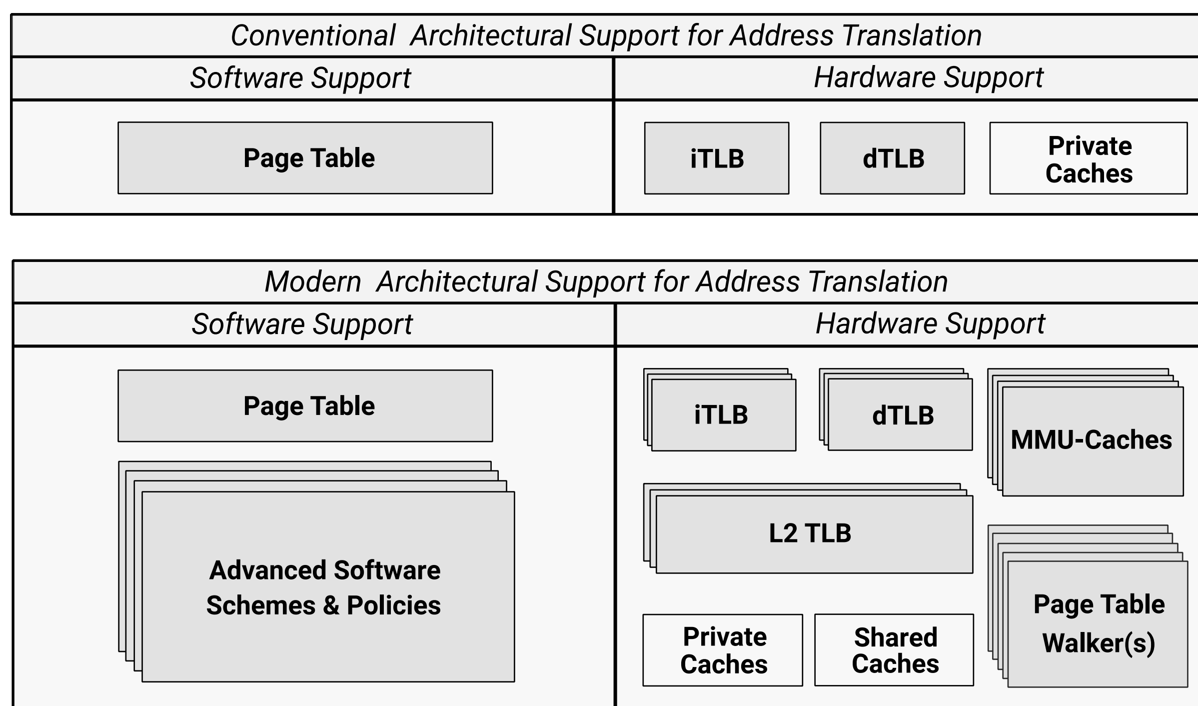


Figure 2.9: Conventional (top) and modern (bottom) software and hardware support for address translation. Modern TLBs have two levels; first-level TLBs are split between data and instructions (dTLB and iTLB, respectively) while L2 TLBs accommodate both data and instruction address translations. Modern designs further split TLBs between different page sizes. MMU-Caches are small cache structures that store entries of the different page table levels. Page table walkers are hardware finite state machines (FSMs) that perform page walks entirely in hardware. Private and shared caches are highlighted in white color since they are not structures dedicated to address translation, but do accommodate PTEs.

### 2.3.3 Architectural Support for Address Translation

Page-based virtual memory provides unique benefits that are vital for the success of computing, as Section 2.3 highlights. However, these benefits come with the latency and energy overheads of performing memory accesses in virtual memory systems; programs operate on virtual addresses, thus upon a memory request the corresponding virtual address needs to be translated into a physical one. To reduce these overheads while avoiding associated performance pathologies, processor vendors are willing to provide both software and hardware support to accelerate address translation [87]; dedicated hardware structures and schemes for the common-case operations and OS policies for the less performance-critical operations and management functions.

Figure 2.9 compares architectural support provided for the uniprocessors of the 80s/90s and modern computing systems of today while breaking it down into support provided by the software and the hardware. In the conventional case (uniprocessors of the 80s/90s), processor vendors employ separate page table caches, known as *Translation Lookaside Buffers or TLBs*, for data and instructions (dTLB and iTLB in Figure 2.9) while storing PTEs into the private caches (*e.g.*, x86-64 architectures implement 64-byte cache lines, thus a single cache line can store 8 PTEs since each PTE is 8 bytes). Furthermore, in the conventional case, both dTLB and iTLB misses are handled by an OS handler that walks the page table to find the requested translation and install it to the corresponding TLB structure (dTLB or iTLB). Finally, architectural support for address translation during the 80s/90s provides support only for standard 4KB pages.

Moving to modern architectural support for address translation, Figure 2.9 shows that there are additional schemes dedicated to address translation compared to the conventional case. On the software side, there are hierarchical page table designs, known as *radix tree page tables*, that efficiently store the virtual to physical mappings of all pages loaded to main memory [269]. Furthermore, there are smart software schemes and policies such as *lazy allocation* [87], *Address Space Layout Randomization (ASLR)* [123, 171, 249], sophisticated *page replacement policies* [91], and intelligent *memory allocators* [173], among others. In addition, there is support for multiple page sizes (*e.g.*, 4KB, 2MB, and 1GB pages in x86-64 architectures) that alleviate some of the pressure placed on the virtual memory subsystem. Literature refers to page sizes larger than base 4KB pages as *superpages* or simply *large pages* [7, 8, 10, 30, 43, 125, 129, 204, 250, 268]. On the hardware side, there are TLB hierarchies with two levels: (i) separate first-level TLBs between instructions and data that are further split into separate TLBs for different page sizes (*e.g.*, 4KB, 2MB, and 1GB pages in x86-64 architectures), and (ii) last-level TLBs that accommodate both instruction and data PTEs that occasionally support multiple page sizes within the same structure. Moreover, there are hardware page table walkers and MMU-Caches. The former are hardware finite state machines (FSMs) that serve TLB misses entirely in hardware, avoiding costly context switches. The latter are small cache structures that store entries from the intermediate levels of the radix tree page table to reduce the memory footprint of page walks. Subsequent sections elaborate on modern software and hardware schemes dedicated to address translation, presented in Figure 2.9.

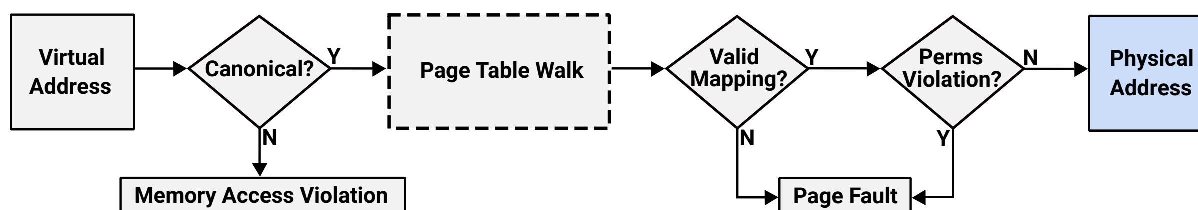


Figure 2.10: Illustration of the address translation process, containing more details than Figure 2.7. Diamonds indicate decision points while dotted lines indicate actions.

#### 2.3.3.1 Page Table

As explained in Section 2.3.2, the *page table* is an OS-managed and architecturally visible structure that accommodates all pairs of virtual-to-physical mappings for each process in the system. The building block of the page table is the *page table entry (PTE)* that contains all relevant information for a virtual page that might be mapped in a physical page.[1] PTEs store information in a compact way to avoid occupying a lot of memory for the page table itself. The information contained in a PTE slightly varies between different architectures. However, typical PTEs contain at least the following information:

- one bit annotating whether or not the virtual page is mapped in a physical page.
- bits that define the permitted operations within the page.
- bits that store information about the cacheability of the page.
- bit(s) indicating whether or not the page was recently accessed.
- one bit annotating whether or not the page has been modified by the core.
- the physical page number that stores the translation of the virtual page.

Figure 2.10 demonstrates how modern systems use the information contained in PTEs by depicting the operation that takes place upon a memory access. First, it is checked whether or not the virtual address is in canonical form; if not, an access violation is triggered. Assuming a canonical virtual address, the next step consists of the page table traversal where the page table is searched for finding the requested translation entry. At the end of the page

[1] A virtual page might be mapped in a physical page since modern OSes map memory *lazily* [87]. Section 2.3.3.5 presents the core idea behind the *lazy allocation* concept.

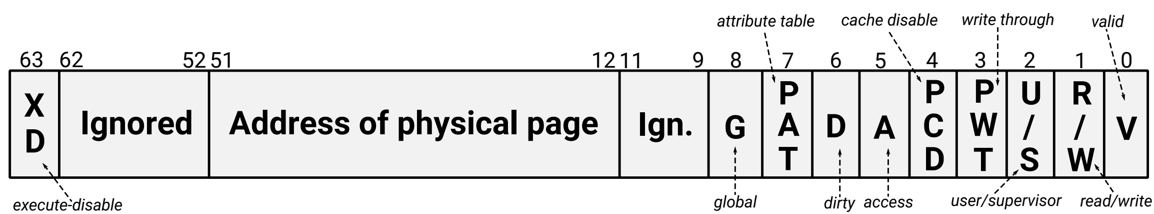


Figure 2.11: PTE format on x86-64 architectures assuming standard 4KB pages.

table walk, the PTE that stores the address translation passes a validation and an access permission check. If any of these checks fails a fault[1] is triggered to indicate that the address translation is not present in the page table. Otherwise, the physical address is returned to the core and the execution continues normally.

Since this thesis is focused on x86-64 architectures, Figure 2.11 depicts the PTE format in x86-64 architectures, assuming standard 4KB pages [31]. Bit 0 reveals whether or not the virtual page is mapped in a physical page frame. Bit 2 indicates whether user code or supervisor code can access the page. Bits 1 and 63 define the write and execute permissions of the page, respectively. Bits 3, 4, and 7 reveal the caching properties of the page. Bits 5 and 6 indicate whether or not the page has been recently accessed and/or written, respectively. Finally, bit 8 indicates whether the page is global or not; global pages have to be invalidated upon context switches.

### Hierarchical Page Tables

In its basic form, the page table is organized as a key-value data structure using the virtual addresses as keys and the physical addresses as values. Such a design is inefficient since it would require multi-gigabytes of memory for just storing the page table, even when applications use only a small part of the virtual address space. For this reason, real-world page table implementations are more agile than a cumbersome key-value data structure design. The fundamental idea behind agile page table designs is the following: only a few applications make use of the entire available virtual address space (this number is even lower

[1]This fault is named *Page Fault*. There are two types of page faults: minor and major. A minor page fault happens when the page is allocated in the virtual address space but not in the physical address space (*e.g.*, due to lazy allocation [87] – Section 2.3.3.5) and typically incurs low latency and energy overheads. A major page fault takes place when the requested data are not present in memory (*e.g.*, swapped out to disk due to limited memory capacity) and need to be fetched from the disk; frequent major page faults deteriorate system performance due to very high latency costs.

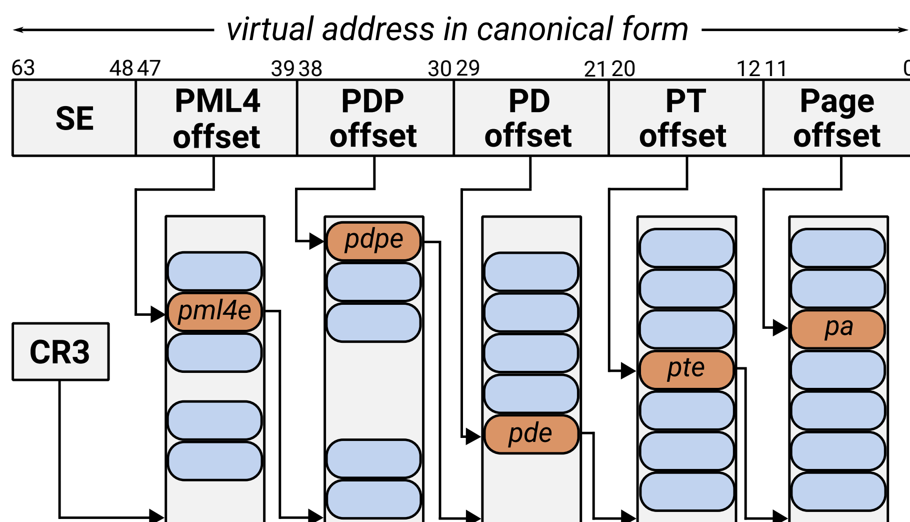


Figure 2.12: Radix tree page table with 4 levels and 48-bit canonical virtual addresses. The names of the page table levels (from left to right) are *Page Map Level 4 (PML4)*, *Page Directory Pointer (PDP)*, *Page Directory (PD)*, and *Page Table (PT)*. Moving to 57-bit canonical virtual addresses would require implementing a radix tree page table with 5 levels.

when page sizes larger than standard 4KB pages are used). In practice, most modern page tables are designed as hierarchical structures, known as *radix tree page tables*. However, there are alternative design approaches. Inverted page tables are such an example and are commonly employed in the PowerPC and UltraSPARC architectures [140, 146, 147]. This thesis targets x86-64 architectures, thus this section focuses on radix tree page tables.

Figure 2.12 presents the design and the operation of a 4-level radix tree page table, assuming standard 4KB pages and virtual addresses in 48-bit canonical form. The names of the page table levels (from left to right) are *Page Map Level 4 (PML4)*, *Page Directory Pointer (PDP)*, *Page Directory (PD)*, and *Page Table (PT)*. To obtain the address translation of a virtual page, all page table levels need to be traversed sequentially. First, the virtual address is split into parts that are used to index the different page table levels; the base address of the first page table level (PML4) is stored in the CR3 register. The operation is the same for all page table levels; the entries of the previous page table level store pointers to the base of the next page table level and the corresponding bits of the virtual address are used for indexing. In this case where the pages are 4KB, the intermediate page table levels (PML4, PDP, and PD) can be interpreted as monolithic page table structures that contain pointers to other monolithic page tables instead of pointers to a physical page.

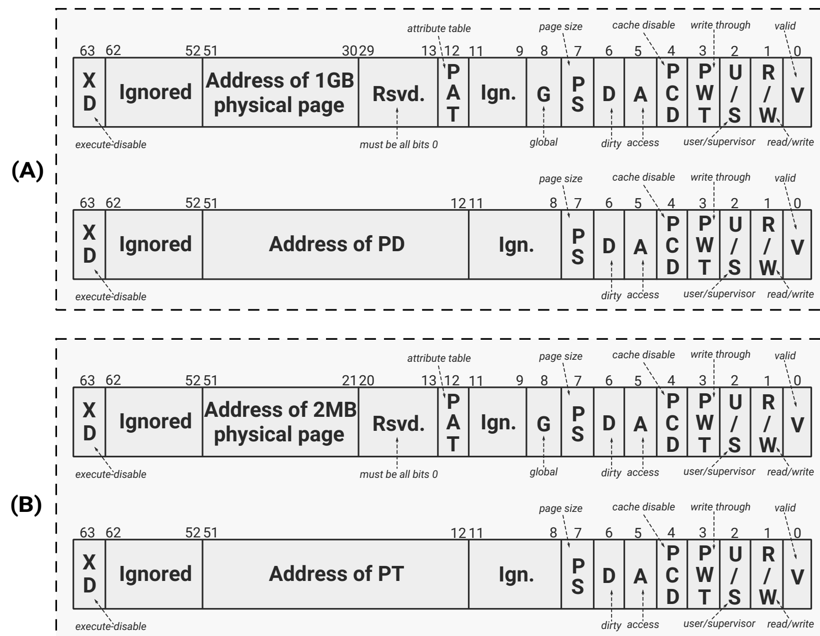


Figure 2.13: (A) Format of a PDP entry that maps to a 1GB page (up) and references a PD (down). (B) Format of a PD entry that maps to a 2MB page (up) and references a PT (down).

### Multiple Page Sizes

The above explained operation takes place when the page size is 4KB. However, contemporary computing systems support multiple page sizes to accelerate address translation. For example, x86-64 architectures provide support for 4KB, 2MB, and 1GB superpages. What is the impact of supporting more than one page sizes on the design and the operation of a 4-level radix tree page table? The answer to this question is that the design of radix tree page tables favors the usage of multiple page sizes since each page table level can be interpreted as an individual page table that may point to another page table or directly to a physical address. To explain the core idea, we assume x86-64 architectures with support for 4KB, 2MB, and 1GB pages. The entries of the PT level of the page table store translation information about a 4KB memory block whereas entries from the PD and PDP levels hold translation information for $2^9 * 4KB = 2MB$ and $2^9 * 2^9 * 4KB = 1GB$ memory areas, respectively.[1][2] Therefore, to support 2MB and 1GB pages alongside standard 4KB pages,

[1]To calculate the memory region mapped by the intermediate page table levels we multiply with $2^9$ since 9 bits of the virtual address are used to index each page table level, as shown in Figure 2.12.
[2]A PML4 entry contains the address translation of a 512GB ($2^9 * 2^9 * 2^9 * 4KB$) memory area, but x86-64 architectures do not use that large page sizes.

x86-64 architectures make the PDP and PD entries store the translations of 2MB and 1GB pages, respectively, instead of storing a pointer to the next page table level. This happens only when the page size is 2MB or 1GB; when the page size is 4KB the PDP and PD levels store pointers to the next page table levels to ensure correct translation for 4KB pages. To deliver such information to the $\mu$architecture implies that the format of the PDP and PD entries would be different when pointing to a physical page or to the next page table level. Figures 2.13 (A, B) present the format of a PDP and a PD entry when pointing to a physical page (1GB and 2MB, respectively) or to the next page table level (PD and PT, respectively). Therefore, page walks for 4KB, 2MB, and 1GB pages incur up to 4, 3, and 2 references to the memory hierarchy (caches, DRAM), respectively. Consequently, superpages reduce the latency and energy cost of page walks due to the requirement of traversing fewer page table levels to obtain the address translation.

#### 2.3.3.2 Translation Lookaside Buffer (TLB)

In virtual memory systems, each memory operation asks for a virtual-to-physical address translation. Focusing on x86-64 architectures with 4-level radix tree page tables, each memory access would require searching for the requested address translation in the radix tree page table. This implies that 4 additional memory references (caches, DRAM) will be triggered, one per page table level, to obtain the requested translation. Then, the processor will perform the actual memory operation that was asked by the program. Introducing 4 additional memory references per memory request would be a major performance obstacle for modern systems, especially when executing applications that frequently access memory.

To reduce the memory references due to page walks and accelerate address translation, processor vendors store the mostly used address translation entries into a dedicated structure named *Translation Lookaside Buffer (TLB)*. More precisely, TLBs are small private per-core buffers that cache entries from the last level of the radix tree page table, presented in Figure 2.12. TLBs partially reduce the address translation overheads by making the most commonly used PTEs rapidly available to the processor, obviating the need for triggering long-latency page table walks to fetch the corresponding translations. Figure 2.14 depicts the process of accessing memory when the system employs a generic TLB structure while illustrating what is typically stored in a TLB entry. In practice, upon a memory access request, the processor looks up the TLB for the requested translation. On TLB hits, the translation is

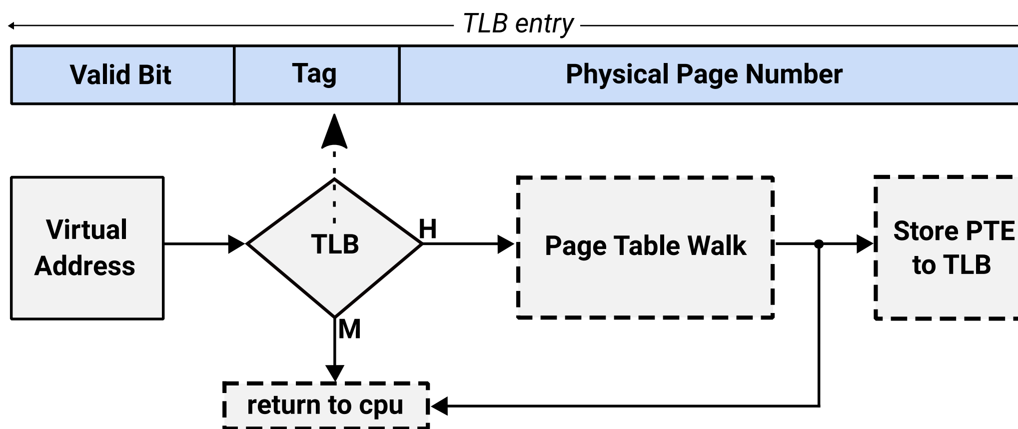


Figure 2.14: The process of address translation when a system employs a generic TLB that stores the most recently used PTEs. Diamonds indicate decision points while dotted lines indicate actions.

returned to the cpu. However, on TLB misses, the page table is traversed to obtain the requested address translation. At the end of the page walk,[1] the corresponding PTE is stored in the TLB for future use.

As explained before, TLBs are set-associative caches dedicated to the last level of the radix tree page table and they are typically placed close to the cpu to ensure that address translations get rapidly available to the cpu. In addition, every memory access needs to go through the TLB, essentially placing the TLB lookup on the critical path of accessing memory. This is the reason why TLBs necessitate fast access times and low look-up latencies. Consequently, TLBs cannot scale to very large sizes and associativities since increasing their size would result in higher access latencies, essentially growing the critical path of memory accesses.

### Hierarchical TLBs

Contemporary applications feature massive working set sizes that place tremendous pressure on the virtual memory subsystem [52, 55, 58, 71, 82, 117, 166, 167, 169, 175, 213, 231, 237, 298]. The increase in applications' memory footprints asks for larger TLBs. Simply increasing the size of a monolithic TLB structure is not a scalable and viable solution to this problem, as explained before. To address the need for larger TLBs, computer architects

[1] Page table walks can be performed either in software or hardware. Section 2.3.3.3 presents both options, highlighting which approach is used by leading processor vendors.

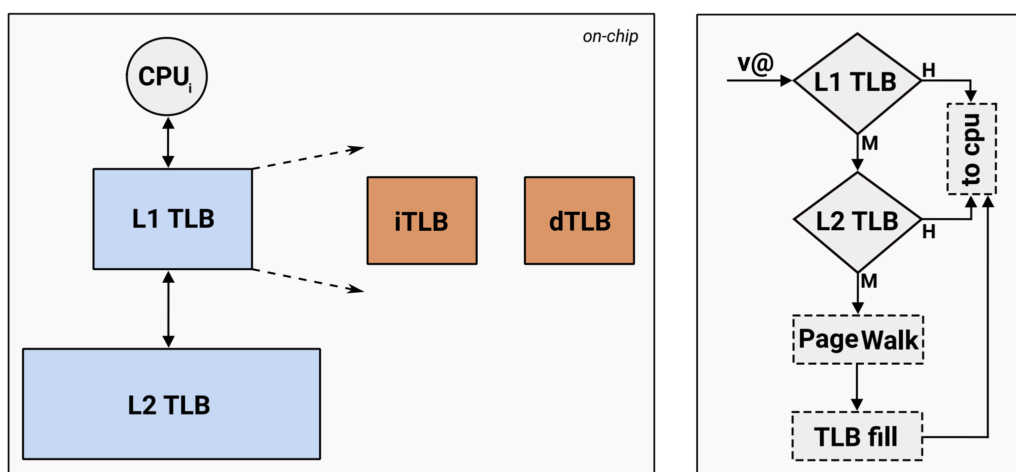


Figure 2.15: TLB hierarchy with two levels (left). L1 TLBs are split between instructions and data. L2 TLBs are larger and slower than L1 TLBs and store both data and instruction PTEs. The flow diagram (right) depicts the process of address translation when a system employs a 2-level TLB hierarchy. Diamonds indicate decision points while dotted lines indicate actions.

design private per-core multi-level TLB hierarchies, similar to hardware cache hierarchies presented in Section 2.1. Figure 2.15 (left) depicts a TLB hierarchy with 2 levels since this is what is implemented in most modern $\mu$architectural designs. The L1 TLB is a small and low-latency structure that ensures fast search for address translations. Moreover, there are separate L1 TLBs for instructions and data (iTLB and dTLB in Figure 2.15, respectively). The employment of separate L1 TLBs for instructions and data appears in almost all modern processor designs because (i) such an approach reduces the chances of pipeline hazards due to port contention in the L1 TLB, (ii) instruction and data references have different locality properties, and (iii) instruction references are more critical for performance since the might cause pipeline stalls while the latency cost of data references can be partially hidden by concepts like out-of-order execution, instruction-level parallelism (ILP), and memory-level parallelism (MLP). Contrarily, L2 TLBs are bigger than L1 TLBs to increase the probability of satisfying as many as possible requests for address translations at the cost of being slower and requiring more area and logic to be implemented than L1 TLBs. An important property of L2 TLBs is that they accommodate both instruction and data address translations.

Figure 2.15 (right) presents the process that takes place upon a memory request when a system employs a 2-level TLB hierarchy. First, the corresponding L1 TLB is searched for the requested translation. On L1 TLB hits, the cpu gets the translation, and the memory request

is replayed [83]. In case the L1 TLB lookup misses, the L2 TLB is probed. On L2 TLB hits, the processor replays the memory access, similar to L1 TLB hits. However, on L2 TLB misses, a page walk is triggered to fetch the translation from the page table. At the end of the corresponding page walk, the requested translation is stored in the TLB hierarchy and the memory request is replayed. Note that the operation of the 'TLB fill' step depends on the TLB inclusion policy, *i.e.*, whether the L2 TLB is inclusive or exclusive of the L1 TLB. The most common case is to implement *mostly inclusive* [86, 297] TLB hierarchies (similar to mostly inclusive hardware caches [152]) where (i) the address translations are filled into both L1 TLB (instruction and data translations are stored into the iTLB and dTLB, respectively) and L2 TLB, and (ii) each TLB level drives evictions based on its own replacement policy, thus there is no need for strict inclusion between L1 TLBs and L2 TLB. For the rest of the thesis, we use TLB to refer to a 2-level TLB hierarchy and dTLB and iTLB to refer to data and instruction L1 TLBs, respectively, unless stated otherwise.

#### Multiple Page Sizes

Despite the existence of multi-level TLB hierarchies, the growth in working set sizes of contemporary applications outpaces the growth in TLB sizes, resulting in frequent TLB misses that deteriorate system performance due to the page walks required for fetching the corresponding address translations. Modern designs increase the effective capacity of the TLB, known as *TLB reach*, by supporting page sizes larger than base 4KB pages (*e.g.*, apart from standard 4KB pages, x86-64 architectures also support 2MB and 1GB superpages). Superpages significantly improve TLB coverage since a single TLB entry for a superpage maps the same memory as multiple TLB entries for standard 4KB pages. For example, a single 2MB page (1GB page) maps the same memory space as 512 (262144) 4KB pages.

From the TLB perspective, handling large pages and exploiting the maximum of their benefits requires additional storage and non-trivial design choices. Separate TLB structures per page size or not? Is the same strategy equally effective for all TLB levels? Leading processor vendors typically employ different L1 TLBs (both iTLB and dTLB) per supported page size since it is difficult to build fast and energy efficient set-associative TLB structures that concurrently support multiple page sizes. Supporting all page sizes within the same structure would require the page size information available at lookup time. However, the page size is known after the TLB lookup. This represents a chicken and egg problem that

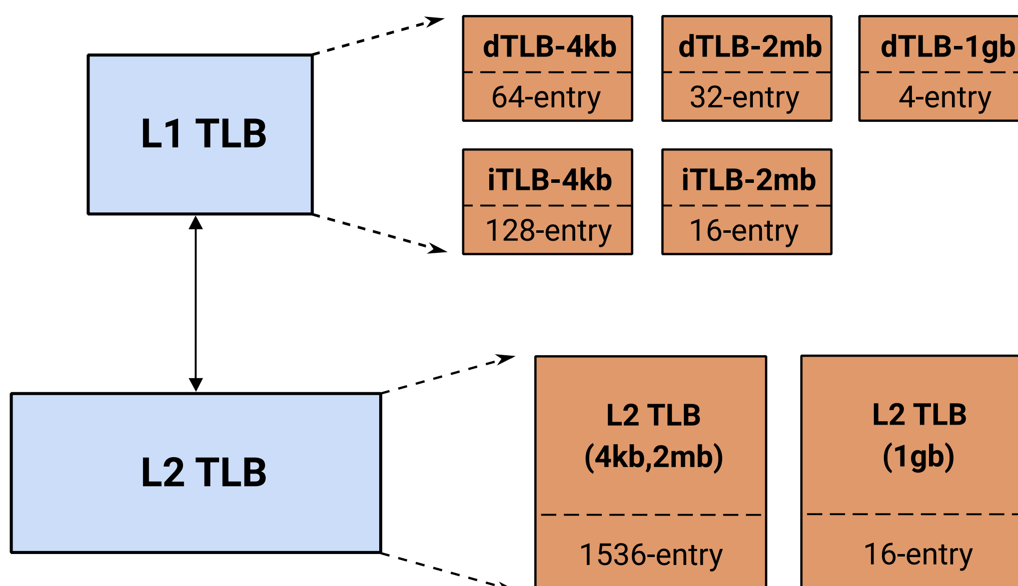


Figure 2.16: TLB organization in modern x86-64 systems that support 4KB, 2MB, and 1GB pages. The capacity of the different TLB structures is taken from Intel's Skylake 2018 chips [48].

processor vendors solve by implementing separate L1 TLBs per page size. Regarding the L2 TLBs, processor vendors follow a different strategy. In practice, they concurrently support a number of page sizes within the same structure while building small separate TLBs for the less used page sizes. For the reasons outlined above, it is challenging to support multiple page sizes within a single set-associative structure. However, vendors do so for L2 TLBs due to the area, storage, and energy overheads of implementing different L2 TLBs for all supported page sizes. We refer interested readers to [48, 87, 215] for more information.

Figure 2.16 shows the TLB organization implemented in recent x86-64 architectures while annotating the number of entries available per TLB structure for Intel's Skylake 2018 chips [48]. There are separate iTLBs and dTLBs that are further split between different page sizes. Typically, x86-64 architectures do not implement separate iTLBs for 1GB pages since applications rarely use 1GB pages for code. The design of L2 TLBs is different; they accommodate both data and instruction translations for 4KB and 2MB pages within a single structure while a different structure is used to store the translations for 1GB pages. Finally, TLB structures that accommodate translations for superpages have fewer entries than the corresponding ones that store translations for 4KB pages because superpages cover much larger memory regions than standard 4KB pages, as explained above.

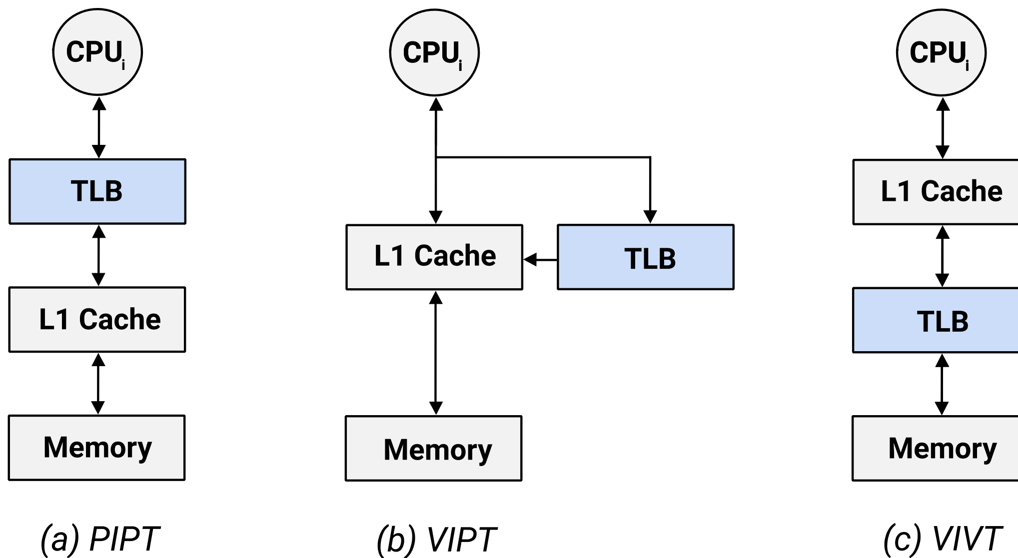


Figure 2.17: TLB placement relative to first-level hardware caches. There are three realistic configurations: (a) Physically Indexed, Physically Tagged (PIPT) caches, (b) Virtually Indexed, Physically Tagged (VIPT) caches, (c) Virtually Indexed, Virtually Tagged (VIVT) caches. Finally, Physically Indexed, Virtually Tagged (PIVT) caches are ignored because they are not used in practice.

#### TLB Placement

The impact of address translation on the overall performance of a system heavily depends on the TLB placement relative to the existing cache hierarchy. Depending on the first-level cache implementation, there are four different organization strategies. However, only three of them could be used in practical designs. Figure 2.17 illustrates these three different configurations.

When the L1 cache is implemented as a *Physically Indexed, Physically Tagged (PIPT)* structure, the address translation needs to be performed before the cache lookup. Such an approach enables simple cache management and eases cache coherence protocols at the cost of placing the entire TLB lookup on the critical path of accessing memory. Therefore, when PIPT L1 caches are used, TLB sizes scale poorly with application needs.

*Virtually Indexed, Physically Tagged (VIPT)* L1 cache implementations permit accessing the TLB and the L1 cache in parallel [70, 274]. This happens because VIPT caches are indexed with bits of the virtual address and are tagged with bits from the physical address. Therefore, the L1 caches can be indexed in parallel with the TLB lookup, partially hiding its latency cost. The drawback of VIPT caches is that they limit the number of L1 sets due

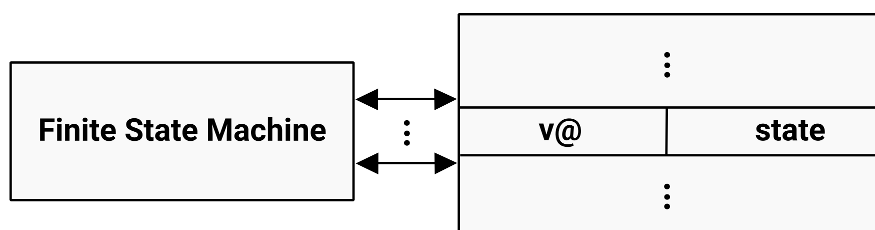


Figure 2.18: The building blocks of a hardware page table walker (PTW) are (i) a finite state machine, and (ii) a buffer storing information about outstanding page walk references to the memory hierarchy (caches, DRAM).

to the requirement of extracting the cache index bits from the page offset. For example, a system with 4KB pages (the page offset is 12 bits) and 64-byte cache lines need to use 6 bits for the cache block offset, thus only 6 bits remain for indexing. Consequently, a system with 4KB pages can support VIPT caches with up to 64 sets.

*Virtually Indexed, Virtually Tagged (VIVT)* caches entirely remove the TLB lookup from the critical path of accessing memory [70, 170, 226, 292, 302]. VIVT caches use the virtual address for both indexing and tagging purposes, permitting TLBs to scale to larger sizes because the TLB lookup needs to be performed after the L1 cache lookup. Despite their latency benefits, VIVT caches have several major drawbacks. It is very challenging to detect and raise access violations, handle synonyms, and support multi-programmed workloads with VIVT caches.

The drawbacks of VIVT caches make processor vendors not include them in real-world $\mu$architectural designs. Commodity processors typically implement VIPT L1 caches and PIPT L2Cs and LLCs [70].

#### 2.3.3.3 Hardware Page Table Walkers

Despite the advent of groundbreaking innovations in TLB management (*e.g.*, hierarchical TLB organizations, support for multiple page sizes), the reach of a modern L2 TLB with 1536 entries (Figure 2.16) and 4KB pages is only 6MB. Contemporary applications have working set sizes much larger than 6MB, thus they place tremendous pressure on the TLB hierarchy, resulting in frequent page walks that incur high latency and energy overheads [52, 55, 58, 63, 64, 71, 82, 84, 94, 97, 115, 117, 160, 161, 162, 163, 166, 167, 169, 175, 213, 231, 237, 276, 298]. Since contemporary memory-intensive applications experience high TLB miss rates, the TLB miss handling mechanism plays a critical role for the overall performance of

the system; effectively handling TLB misses can reduce the address translation performance bottleneck.

Until the early 2000s, page table walking was performed in software [85, 93, 145, 149, 151, 201]. In software-managed approaches, every TLB miss causes a context switch and an OS handler performs the page walk. At the end of each page walk, the translation is stored in the TLB via dedicated ISA instruction. Despite the simplicity and flexibility offered by software-based page walking, it incurs high performance penalties due to the requirement of context switching to the OS on every TLB miss that forces the pipeline to be flushed while polluting the content of the $\mu$architectural structures.

Modern architectures perform page walks entirely in hardware to obviate the need for context switching upon TLB misses [67, 71, 86, 120]. To do so, vendors employ private per-core hardware *page table walkers (PTWs)*. A hardware PTW is a finite state machine that is aware of the page table organization coupled with a buffer that keeps information about the outstanding TLB misses, similar to the Miss Status Handling Registers (MSHRs) for the hardware caches. Figure 2.18 depicts the building blocks of a hardware PTW; a finite state machine and a dedicated buffer.

The operation of a hardware PTW is straightforward. Upon a TLB miss, the hardware PTW accesses the CR3 register that stores the base address of the first page table level (PML4 in x86-64 architectures) and performs the page walk, as explained in Section 2.3.3.1, while keeping all necessary information in the PTW buffer to ensure that only outstanding page walk memory references will be issued.

Hardware PTWs significantly improve the performance of virtual memory systems for a number of reasons. First, they obviate the need for pipeline flushes while avoiding cache pollution and costly interrupts upon TLB misses since they operate in hardware without any OS involvement. In addition, hardware PTWs offer the opportunity to overlap the page walk execution with useful work and (partially) hide its latency overheads. The only disadvantage of hardware PTWs is that they are cumbersome with respect to the page table organization, meaning that they are designed to operate with a specific page table organization. However, the benefits offered by hardware PTWs make leading processor vendors ignore their limited flexibility while integrating multiple hardware PTWs (up to four) per each core of the system to permit the execution of multiple page walks in tandem [190].

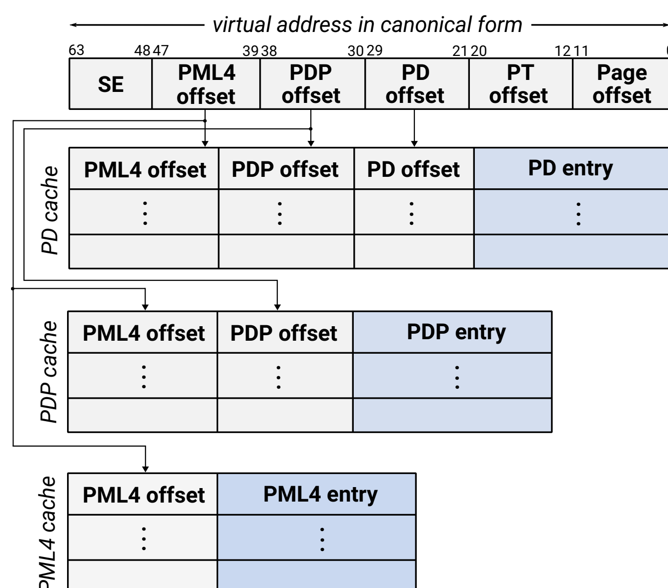


Figure 2.19: MMU-Caches implemented in Intel x86-64 architectures.

#### 2.3.3.4 MMU-Caches

Despite the advent of hardware PTWs, page walks still incur great latency and energy overheads, especially for applications that exacerbate TLB pressure. To minimize the latency overheads of page walks upon TLB misses, computer architects have invented the *MMU-Caches* [66, 82, 142, 215]. The MMU-Caches are small and fast caches that store entries from the intermediate page table levels (PML4, PDP, and PD in x86-64 architectures), similar to TLBs that are caches for the last page table level, as explained in Section 2.3.3.2.

Two main reasons motivated the integration of MMU-Caches in almost all modern virtual memory systems. First, a page table walk triggers one memory reference per page table level, as explained in Section 2.3.3.1. Hitting in one of the MMU-Caches eliminates one of these memory references; at the extreme, hitting in all MMU-Caches implies that the corresponding page walk would require only one memory reference for the last page table level to obtain the address translation. The second reason is that MMU-Caches have very low storage and area costs because the intermediate page table levels map larger portions of the address space than the leaf page table level. For example, in an x86-64 paging scheme, a PML4, PDP, PD, and PT entry covers a 512GB, 1GB, 2MB, and 4KB memory region, respectively. Therefore, a few entries are adequate for MMU-Caches to cover a big portion of the address space and enjoy high hit rates.

Different processor vendors implement different MMU-Cache variations. Intel implements a variation named *Page Structure Caches (PSCs)* while AMD uses the *Page Walk Caches (PWCs)*. The former are tagged with bits from the virtual address and permit parallel lookups while the latter are PIPT caches that need to be looked up serially. Figure 2.19 presents the MMU-Caches implemented in Intel x86-64 processors. The PLM4 cache, PDP cache, and PD cache are indexed with the PML4, PML4+PDP, and PML4+PDP+PD offset bits from the virtual address, respectively. In practice, upon a TLB miss, all MMU-Caches are scanned in parallel and the hit with the longest index bits (if any) is used as starting point for the page walk. For example, a hit in the PD cache would skip the PML4, PDP, and PD lookups, thus the corresponding page walk would trigger a single reference to the memory hierarchy (caches, DRAM) for the leaf level (PT) to obtain the address translation.

#### 2.3.3.5 Software Schemes and Policies

**Lazy Allocation** [87] is a technique for managing new virtual memory allocations. Instead of immediately mapping all virtual pages within a virtual memory area to new physical pages, lazy allocation performs the allocations when the program accesses the corresponding virtual pages for the first time. The main benefit of this technique is that there is no memory waste since it makes sure that a virtual page is actually used before performing the physical memory allocation.

**Copy-On-Write** [222, 244] works similarly to the lazy allocation scheme; the difference is that it targets page duplication and not new virtual memory allocations. In practice, when a memory region is duplicated (*e.g.*, due to a *fork* system call) the new region is not physically allocated to a new region until there is a modification to one of the two pages in question.

**Address Space Layout Randomization (ASLR)** [123, 171, 249] is a scheme aimed at protecting virtual memory systems from security attacks. ASLR schemes randomly arrange the address space positions of important data areas to prevent adversaries from predicting processes' target addresses. ASLR variations have been adopted in most modern OSes. Recently, Linux has also employed *Kernel Address Space Layout Randomization (KASLR)* [33, 154], a scheme that leverages ASLR to randomize the positions of kernel pages.

**Page Replacement Policy** aims at predicting how far in the future a page will be accessed and evict pages from main memory that will be used furthest ahead in the future. The most commonly used page replacement policy in systems is the *Least Recently Used (LRU)* replacement policy [106] or a variation of it. In practice, most OSes implement *pseudo-LRU* [50, 106], a policy that obviates the need for storing timestamps per allocated page by maintaining a linked list of pages. The most common implementation of the pseudo-LRU algorithm is the CLOCK algorithm [91]. CLOCK leverages the PTE access bits, presented in Section 2.3.3.1, to get an approximation of the working set of a process. To avoid starvation, CLOCK makes sure that the access bits of all pages are periodically reset.

**Memory Allocators** are schemes that try to effectively allocate physical memory upon memory requests. Intelligent memory allocators have to gracefully deal with two types of memory fragmentation [77, 78, 299]; external and internal fragmentation [27]. External fragmentation refers to memory holes that are not reusable due to their small size. Internal fragmentation refers to the wasted space within a single allocation unit. The literature presents simple memory allocators like Best Fit, First Fit, and Worst Fit, among others [26]. However, all these allocators have major drawbacks that hinder their integration into real-world schemes. Modern OSes like Linux employ buddy allocators [173]. Such allocators try to minimize both internal and external fragmentation while performing fast allocations and deallocations of memory. We refer interested readers to [87] for a complete explanation of buddy allocation and its properties.

## 2.4 Hardware Prefetching in Virtual Memory Systems

This section highlights that hardware prefetching can be also applied for the $\mu$architectural structures that are part of the virtual memory subsystem (*e.g.*, TLBs), elaborates on the properties of hardware prefetching for the TLB hierarchy, and highlights the impact of virtual memory on hardware prefetching applied to the different levels of the cache hierarchy. To do so, it considers a generic system with the most commonly employed $\mu$architectural organization, presented in Figure 2.20; a 2-level TLB hierarchy and a 3-level cache hierarchy where L1 caches are implemented as VIPT structures and L2C and LLC are PIPT structures, as explained in Section 2.3.3.

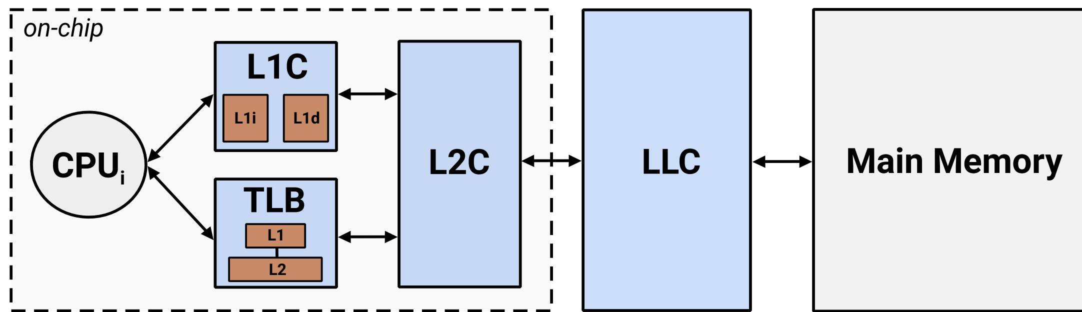


Figure 2.20: Cartoon depicting the $\mu$architecture of a system that employs a two-level TLB hierarchy, VIPT L1 caches and PIPT lower-level caches.

### 2.4.1 Hardware Prefetching for the Virtual Memory Subsystem

Virtual memory makes the Memory Wall 'taller' because memory requests that miss in the TLB hierarchy trigger page walks that traverse the memory hierarchy (hardware caches and main memory) potentially multiple times to fetch the requested translation, as explained in Section 2.3.3.1. In other words, virtual memory exacerbates the Memory Wall bottleneck.

Despite innovations in architectural support for accelerating address translation, presented in Sections 2.3.3, the advent of emerging workloads, spanning from HPC to server and datacenter applications, that feature massive data and code working set sizes place tremendous pressure on the existing TLB hierarchies, resulting in high TLB miss rates that harm the overall performance of the system.

#### 2.4.1.1 TLB Prefetching

Hardware prefetching can be also applied to the TLB hierarchy since TLBs are set-associative $\mu$architectural structures, similar to hardware caches. Effective hardware TLB prefetching has the potential to mitigate the latency cost of TLB misses due to its intrinsic properties:

- TLB prefetching is a pure $\mu$architectural technique that relies only on the memory accesses patterns of the application.
- TLB prefetching is independent of the system state (load, fragmentation).
- TLB prefetching does not disrupt the existing virtual memory subsystem.
- TLB prefetching does not require any OS involvement

| virtual page number (vpn) | physical page number (ppn) | attribute bits |
|---|---|---|

Figure 2.21: Prefetch Buffer (PB) entry.

Intuitively, hardware prefetching can be applied for all TLB levels. In practice, computer architects opt to apply prefetching only for the L2 TLB for two reasons. First, L1 TLB misses might result in L2 TLB hits, thus their latency cost might be only the L2 TLB lookup. Secondly, implementing prefetching at the L1 TLB level may result in performance and timeliness issues as L1 TLBs are usually designed to be small in size to minimize access times. On the other hand, L2 TLB misses trigger page walks that inject additional references to the memory hierarchy to obtain the requested translation, incurring high latency and energy overheads. Finally, for the rest of the thesis, we use *TLB prefetcher* to refer to an *L2 TLB prefetcher*, unless stated otherwise.

### TLB Prefetching Properties

TLB prefetchers should address the *$W^3$ challenge*, presented in Section 2.2.2, similar to cache prefetchers. Although prior TLB prefetchers [36, 60, 85, 165] use various prefetching algorithms to decide what to prefetch, they uniformly address the other two challenges; where to prefetch and what to prefetch. They use a small buffer, named *Prefetch Buffer (PB)*,[1] to store the prefetched PTEs since it has been shown that placing prefetched PTEs directly into the TLB hierarchy might pollute the TLB content when TLB prefetching is inaccurate [36, 85, 165]. Figure 2.21 presents the content of a PB entry; the virtual page number, the physical page number, and attribute bits presented in Section 2.3.3.1. As opposed to cache prefetchers that issue prefetches on every cache access, TLB prefetchers generate prefetch requests upon TLB misses and not on every TLB access since prefetching at every TLB access might initiate prefetch requests for pages already stored in the TLB.

### TLB Prefetching in Practice

Before diving into the operation of a system that employs hardware TLB prefetching, we present TLB prefetching terminology that is heavily used throughout the rest of the thesis.

[1] PBs typically have up to 64 entries to ensure fast lookup times and use the LRU replacement policy.

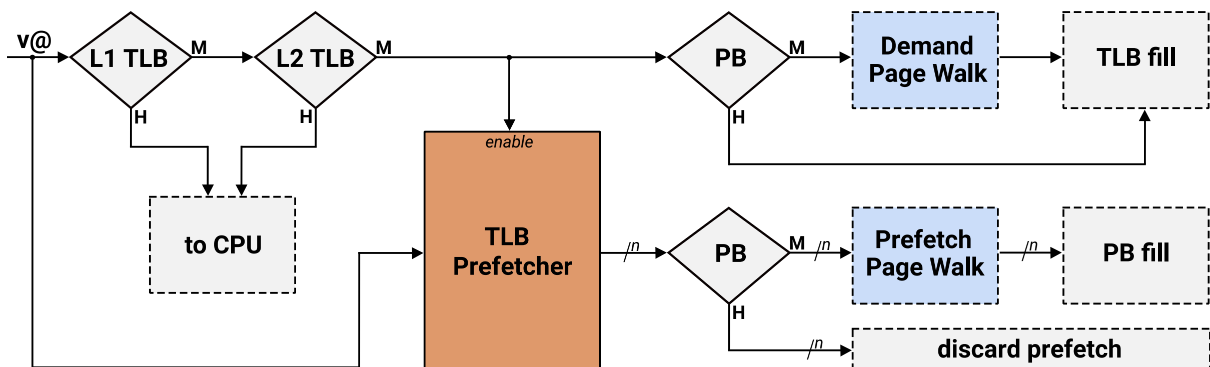

Figure 2.22: Operation of a system that employs a generic hardware TLB prefetcher. Diamonds indicate decision points while dotted lines indicate actions.

**Demand page walk** refers to a page walk triggered due to a memory request issued by a core. When TLB prefetching is not applied, all page walks are demand page walks.

**Prefetch page walk** refers to a page walk triggered to speculatively fetch an address translation from the page table. Prefetch page walks are performed in the background without stalling the pipeline execution. TLB prefetchers issue prefetch page walks to fetch address translations from the page table.

**Page walk memory reference** refers to a reference due to a page walk (demand or prefetch) that is served by the memory hierarchy (L1⟷L2⟷LLC⟷DRAM). Page walk memory references are triggered for accesses that miss in the MMU-Caches, as described in Section 2.3.3.4. Therefore, a page walk might introduce up to 4 memory references and not less than 1 memory reference, assuming a 4-level page table.

Figure 2.22 depicts the operation of a system that employs a generic TLB prefetcher, considering the most common scenario whereby a PB is used to store the prefetched PTEs and the prefetch logic is engaged on L2 TLB misses. When a memory access occurs, the L1 TLB is initially looked up and, on a miss, the L2 TLB is probed. In case the L2 TLB lookup results in a miss, the requested PTE is searched for in the PB. If the translation is present in the PB, it is moved to the TLB hierarchy, the page walk is avoided, and the processor replays the memory request. However, on a PB miss, a *demand page walk* is initiated to fetch the corresponding address translation from the page table. In case of either PB hit or miss, the

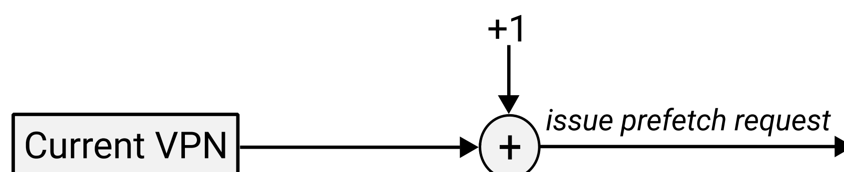


Figure 2.23: SP's basic operation.

TLB prefetcher is activated and produces new prefetch requests based on its prefetching algorithm. Prefetch requests for address translations that are not already stored in the PB,[1] require separate *prefetch page walks* to fetch the corresponding translations from the page table and store them in the PB. Notably, TLB prefetchers permit only non-faulting prefetches since prefetches are speculative events. Finally, the above explained operation is identical for both instruction or data memory accesses; a TLB prefetcher might target to prefetch for the instruction TLB miss stream or the data TLB miss stream.

#### 2.4.1.2 Previously Proposed TLB Prefetchers

Processor vendors do not publicly disclose whether they apply TLB prefetching or not. However, there are academic works that propose different data TLB prefetchers. The most important ones are presented below. Note that there is no previously proposed TLB prefetcher for instruction accesses due to the historically small instruction footprints of applications.

**Sequential Prefetcher (SP)**

SP [165, 277] prefetches the PTE located next to the one that triggered the TLB miss, as shown in Figure 2.23, similar to the next-line cache prefetcher [105, 260].

**Arbitrary Stride Prefetcher (ASP)**

ASP [60, 165] is a TLB prefetcher that targets varying stride TLB miss patterns. To do so, it uses a prediction table indexed with the PC. Figure 2.24 shows the organization of ASP's prediction table as well as its operation upon prediction table hits in steps. Each prediction table entry has four fields: the PC for indexing, the previous virtual page that caused a TLB

[1] TLB prefetchers check whether the translations of their prefetch requests already reside in the PB, but not in the TLB hierarchy because searching the TLB hierarchy for duplicates would contend with demand TLB accesses, potentially delaying the latter.

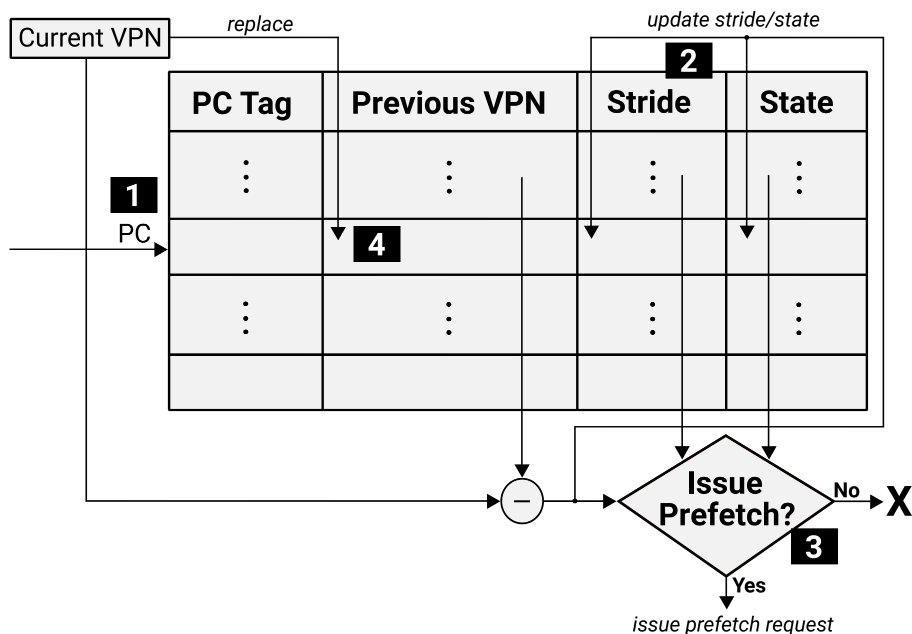


Figure 2.24: ASP's operation upon prediction table hits. Diamonds indicate decision points.

miss while accessed by that PC, the corresponding stride, and a state describing whether the stride has been unchanged for at least two consecutive prediction table hits.

The operation of ASP is simple. On a TLB miss, ASP looks up the prediction table for possible hits (step 1 in Figure 2.24). On a prediction table miss, the PC is stored in the first field of a prediction table entry, the stride field is invalided, and the counter of the state field is reset. On a table hit, ASP updates the stride and the state fields using the current and previous missing virtual pages 2. If there is no change in the stride field, the counter of the state field is increased; otherwise, it is reset. A prefetch takes place only when the counter of the state field is greater than two 3. Finally, in case of either table hit or miss, the current virtual page number is stored in the second field of the corresponding entry 4.

#### Distance Prefetcher (DP)

DP [165] is a TLB prefetcher that correlates miss patterns with distances between virtual pages that produce consecutive TLB misses. To do so, DP leverages a prediction table. Each prediction table entry consists of three fields: the corresponding distance for indexing, and two predicted distances. Figure 2.25 presents (i) the organization of DP's prediction table, and (ii) the operation of DP upon a prediction table hit in steps.

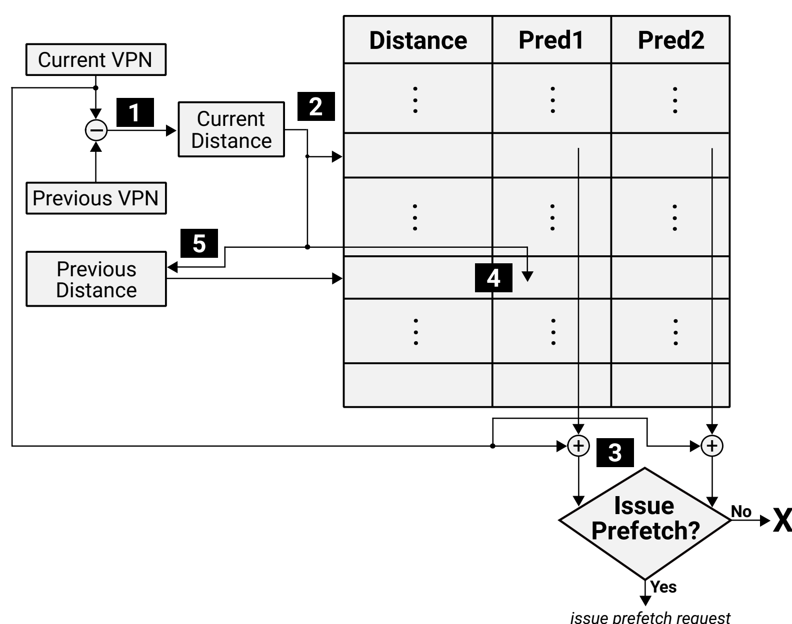


Figure 2.25: DP's operation upon prediction table hits. Diamonds indicate decision points.

On a TLB miss, DP initially computes the signed distance between the current and the previous missing virtual pages and indexes the prediction table (steps 1 and 2 in Figure 2.25). On prediction table hits, DP issues two prefetches using the current missing virtual page number and the predicted distances contained in the second and the third fields of the hit entry 3. Otherwise, a new entry is inserted into the prediction table with all prediction slots empty. In case of either table hit or miss, the entry corresponding to the previous TLB miss is updated by inserting the distance between the current and previous missing virtual pages in the least recently used prediction slot 4. Finally, the currently computed distance is stored in the register holding the previous distance 5.

### Markov Prefetcher (MP)

MP [165] targets irregular TLB miss patterns by building Markov chains out of the TLB miss stream. Like ASP and DP, MP also leverages a prediction table to drive prefetching decisions. Each prediction table entry has three fields: the virtual page number for indexing, and two prediction slots that store the virtual page numbers of the PTEs to be prefetched when a new TLB miss occurs on the virtual page stored in the first field. Figure 2.26 depicts MP's prediction table as well as its basic operation upon a prediction table hit.

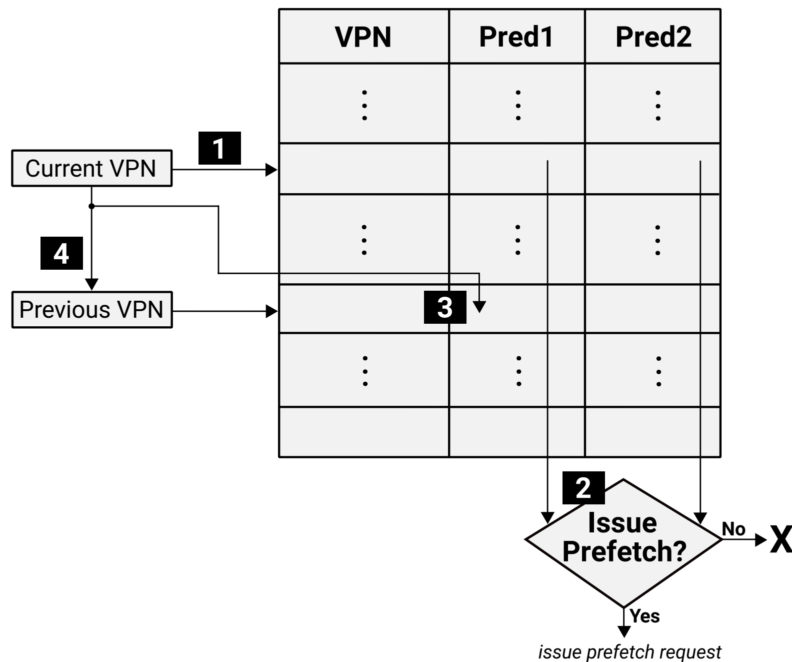


Figure 2.26: MP's operation upon prediction table hits. Diamonds indicate decision points.

Upon a TLB miss, MP indexes its prediction table using the currently missed virtual page (step 1 in Figure 2.26). On prediction table hits, a separate prefetch request is issued for each one of the prediction slots of the hit entry 2. On prediction table misses, there is no prefetching happening for the current TLB miss, and a new entry is allocated to accommodate the currently missed virtual page with all prediction slots marked as invalid. In either case of prediction table hit or miss, the currently missed virtual page is stored into one of the prediction slots of the entry that accommodates the previously missed virtual page 3. Finally, the virtual page that currently produced a TLB miss is stored in the register holding the previous virtual page 4.

### Additional TLB Prefetching Schemes

The previously presented TLB prefetchers (SP, ASP, DP, and MP) are state-of-the-art TLB prefetching mechanisms. However, there are additional works in the domain of TLB prefetching. Bhattacharjee *et al.* [85] propose two TLB prefetching schemes for parallel applications. The first exploits TLB misses on common virtual pages among cores and pushes TLB entries from the leader core to other cores. The second is based on DP and exploits distance-

predictable TLB miss patterns across cores. On the software side, there is the Recency-based TLB Preloading scheme [243]. This is a software mechanism that modifies the page table so that each PTE stores the virtual page that is subsequently accessed and drives prefetching by previously observed access pattern; a fundamentally similar approach that only requires $\mu$architectural modifications is the MP prefetcher. The main insight behind this work is that pages that were used recently will be referenced again.

### 2.4.2 Cache Prefetching in Virtual Memory Systems

In virtual memory systems, a hardware cache can be indexed and/or tagged with virtual or physical addresses, as explained in Section 2.3.3.2. The cache implementation determines whether prefetching is driven with virtual or physical addresses. The prefetchers placed alongside VIPT caches have the opportunity to drive prefetching decisions using virtual addresses whereas the prefetchers of PIPT caches are obliged to use physical addresses to generate prefetch requests. To demonstrate the impact of virtual memory on hardware cache prefetching, this section considers the most common scenario where the L1Cs are implemented as VIPT structures and the lower-level caches (L2C, LLC) are PIPT structures, as presented in Section 2.4 and Figure 2.20.

#### Prefetching at L1C

Modern first-level cache prefetchers use virtual addresses to generate prefetches since L1Cs are indexed with virtual addresses. Driving prefetching decisions with virtual addresses offers the following benefit. Two addresses that are contiguous in the virtual address space might be very distant in the physical address space. In such cases, the access patterns are fairly easy to detect in the virtual address space but nearly impossible to detect in the physical address space. However, this is not the only way virtual memory impacts VIPT L1Cs. Modern systems provide architectural support for multiple page sizes, not only standard 4KB pages. For example, x86-64 architectures concurrently support 4KB, 2MB, and 1GB pages, as explained in Section 2.3. Prefetchers placed alongside the L1C have direct access to the TLB hierarchy. Consequently, L1C prefetchers are not obliged to prefetch within standard 4KB page boundaries since they can access the TLB hierarchy to extract the virtual-to-physical mappings of the pages where the prefetched blocks reside. Conceptually, crossing 4KB page boundaries for prefetching at L1C is a straightforward concept.

However, things are more complicated in practice. What should L1C prefetchers do when the translation of the page where the prefetched block resides is not present in the TLB? Should they discard the prefetch request or fetch the corresponding translation from memory? Doing so would definitely impact (positively or negatively) performance, bandwidth, and energy consumption depending on the accuracy of the page-crossing prefetching. Another critical metric that would be greatly impacted by page-crossing prefetching is the timeliness of prefetching; even if page-crossing prefetching is accurate, it might negatively impact prefetching timeliness and the system's overall performance when it reaches DRAM to find the address translations since L1C prefetchers require quick turnaround times on memory accesses due to the sheer amount of requests seen at the first-level caches. For these reasons, state-of-the-art L1C prefetchers [208] typically permit prefetching within standard 4KB page boundaries.

**Prefetching at L2C/LLC**

Things are less complicated for the lower-level cache prefetchers (L2C and LLC prefetchers) because virtual addresses are not propagated to these caches since they are implemented as PIPT structures. Therefore, lower-level cache prefetchers are forced drive prefetching using physical addresses and they typically do not cross 4KB physical page boundaries since physical address contiguity is not guaranteed, *i.e.*, contiguous virtual addresses might not be contiguous in the physical address space, thus permitting prefetching beyond 4KB physical boundaries is susceptible to security issues. Allowing lower-level prefetchers to speculatively cross 4KB physical page boundaries may result in prefetching data from pages that a given process does not have access to, thereby opening a side channel in which an adversary can detect if the victim has accessed a page despite the adversary not having permissions to that page [96, 128, 287]. Indeed, a recent reverse engineering study [287] demonstrates how to perform a side-channel attack on recent Apple processors (*e.g.*, Apple M1, M1 Max, M1 Pro) by exploiting page-crossing prefetching at the lower-level caches.

# 3 Agile Data TLB Prefetching

## 3.1 Introduction

Address translation associated with data accesses is a major performance and energy bottleneck in workloads featuring large data working sets and low memory reference locality [55, 58, 66, 71, 82, 99, 117, 166, 175, 202, 237, 298] due to the requirement for traversing the page table to find the corresponding address translation entries. Page walk references to the memory hierarchy incur high latency overheads, as explained in Section 2.3.3.1, aggravating the Memory Wall bottleneck.

Prior work has quantified the cost of TLB performance [55, 71, 82, 237] and has proposed several approaches to mitigate the overheads of address translation associated with data accesses. These approaches mainly fall into three categories:

- Increasing the effective capacity of the TLB, *i.e.*, TLB reach, by introducing explicit hardware and OS support for address translation [71, 167, 169, 183, 213, 219, 221].

- Reducing the latency cost of TLB misses [52, 65, 66, 82, 241, 259, 300].

- Reducing the number of TLB misses by prefetching PTEs ahead of demand TLB accesses [85, 165, 188, 243].

This chapter focuses on the last category, TLB prefetching, to reduce the address translation overheads associated with data accesses due to its unique properties; a TLB prefetcher operates entirely at the $\mu$architecture without any OS involvement, is independent of the system state, relies only on the memory access pattern of the application, and does not disrupt the existing virtual memory subsystem.

Prior data TLB prefetchers, presented in Section 2.4.1.2, always trigger prefetch page walks in the background to prefetch PTEs. Since page walks introduce additional references to the memory hierarchy, data TLB prefetching might harm performance if the prefetched PTEs are not consumed by future demand TLB accesses. Although TLB prefetching has the potential to reduce the number of TLB misses, the large number of additional memory references it triggers can undermine its potential for performance improvement and increase the overall energy consumption of the system.

This chapter exploits the locality of PTEs in the last level of the radix tree page table, presented in Section 2.3.3.1, to improve the performance of TLB prefetching while reducing its cost in terms of page walk references to the memory hierarchy and energy consumption of address translation. The core idea behind page table locality is that contiguous PTEs are stored within the same cache line at the end of each page walk because the PTEs are 8 bytes while cache lines are 64 bytes in x86-64 architectures. Therefore, a single 64-byte cache line can accommodate up to 8 contiguously-stored PTEs that can be prefetched 'for free'. Prior work leverages page table locality [86, 219, 221, 255] to increase the effective capacity of TLBs and reduce the number of page walks, but does not exploit it for TLB prefetching purposes. This work leverages page table locality to enhance the performance and reduce the cost of TLB prefetching in terms of page walk references. We demonstrate that naively prefetching all neighboring PTEs into a TLB buffer after a page walk results in suboptimal performance gains. In response, we propose *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that predicts through sampling which of these neighboring PTEs are more likely to prevent future TLB misses and fetches them into a dedicated TLB buffer, named Prefetch Buffer (PB), as explained in Section 2.4.1.1. We highlight that SBFP can be combined with any TLB prefetcher to achieve notable performance gains while reducing the memory footprint of page walks and the energy consumption of address translation.

Moreover, this work proposes the *Agile TLB Prefetcher (ATP)*, a composite data TLB prefetching scheme particularly designed to exploit the benefits of the SBFP mechanism. The design of ATP is driven by our analysis findings which indicate that no single prior data TLB prefetcher performs best among different types of applications, and some workloads do not benefit from TLB prefetching due to irregular patterns. Unlike state-of-the-art data TLB prefetchers that correlate patterns with only one feature (*e.g.,* constant strides, PC, distances between virtual pages that produce consecutive data TLB misses), ATP combines three low-cost data TLB prefetchers and adapts its prefetching strategy depending on the memory access pattern of the application. To do so, ATP relies on two low-cost mechanisms: (i) selection logic that dynamically activates the most appropriate data TLB prefetcher in terms of both the accuracy of the prefetched PTE and also the usefulness of its corresponding free prefetches selected by the SBFP scheme, and (ii) an adaptive throttling mechanism that disables TLB prefetching during phases that do not benefit from it.

In summary, this chapter makes the following contributions:

- We evaluate the state-of-the-art data TLB prefetchers, presented in Section 2.4.1.2, using a large set of both industrial and academic workloads provided by Qualcomm for the 1$^{st}$ Contest on Value Prediction (CVP-1) [19], the SPEC CPU 2006 [136] and SPEC CPU 2017 [42] benchmark suites, the GAP benchmark suite [74], and the XS-Bench benchmark [45].

- We propose *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme that exploits page table locality by predicting which of the adjacent PTEs present in a 64-byte cache line are most likely to save future data TLB misses and prefetch them into a dedicated TLB buffer. We demonstrate that combining SBFP with new and state-of-the-art data TLB prefetchers provides significant performance and energy benefits.

- We propose *Agile TLB Prefetcher (ATP)*, a composite data TLB prefetcher that correlates patterns with multiple features by combining three easily implementable and low-scope data TLB prefetchers while maximizing the impact of SBFP. ATP introduces adaptive selection and throttling mechanisms to enable the most appropriate of its constituent TLB prefetchers per TLB miss while disabling TLB prefetching during phases that it is not helpful.

- We propose a unified solution that combines ATP and SBFP. This approach yields a geometric mean speedup of 16.2%, 11.1%, and 11.8% with 37%, 26%, and 5% average reduction of page walk references to the memory hierarchy for the Qualcomm, SPEC, and Big Data (GAP+XSBench) workloads over a baseline without data TLB prefetching, respectively. Over the best state-of-the-art data TLB prefetcher for each benchmark suite, ATP coupled with SBFP improves performance by 8.7%, 3.4%, and 4.2% for the Qualcomm, SPEC, and Big Data workloads, respectively.

## 3.2 Background

All necessary background information about architectural support for virtual memory systems and hardware TLB prefetching are presented in Sections 2.3.3 and 2.4, respectively. This section builds on top of the concepts presented in Sections 2.3.3 and 2.4, thus understanding these concepts is a prerequisite for following the rest of this chapter.

This chapter focuses on x86-64 architectures and considers a system with a 2-level TLB hierarchy, a radix tree page table with 4 levels, MMU-Caches with 3 levels (called *Page Structure Caches (PSCs)* in x86-64 architectures), and a 3-level cache hierarchy, similar to Figure 2.20 in Section 2.4. In addition, it considers the most common scenario, described in detail in Section 2.4, where TLB prefetching is solely applied for the L2 TLB, TLB prefetches are placed into a dedicated TLB buffer, named Prefetch Buffer (PB), and the TLB prefetcher is activated upon L2 TLB misses for data accesses; there is no prefetching happening upon L2 TLB misses for instruction references. For the rest of this chapter, we use TLB prefetcher to refer to an L2 TLB prefetcher for data accesses, unless stated otherwise.

### 3.2.1 Page Table Locality

In x86-64 architectures, the cache line size is 64 bytes and PTEs are stored contiguously in memory while occupying precisely 8 bytes each. Therefore, a single 64-byte cache line can accommodate up to 8 contiguously-stored PTEs [86, 278]. In practice, when a requested PTE is read from memory at the end of a page walk, it is grouped with 7 neighboring PTEs and they are stored into a single 64-byte cache line. Consequently, a cache line holds the requested address translation plus 7 more PTEs that do not require additional accesses to the memory hierarchy to be prefetched; we refer to these PTEs as ‘free’ prefetches. Note

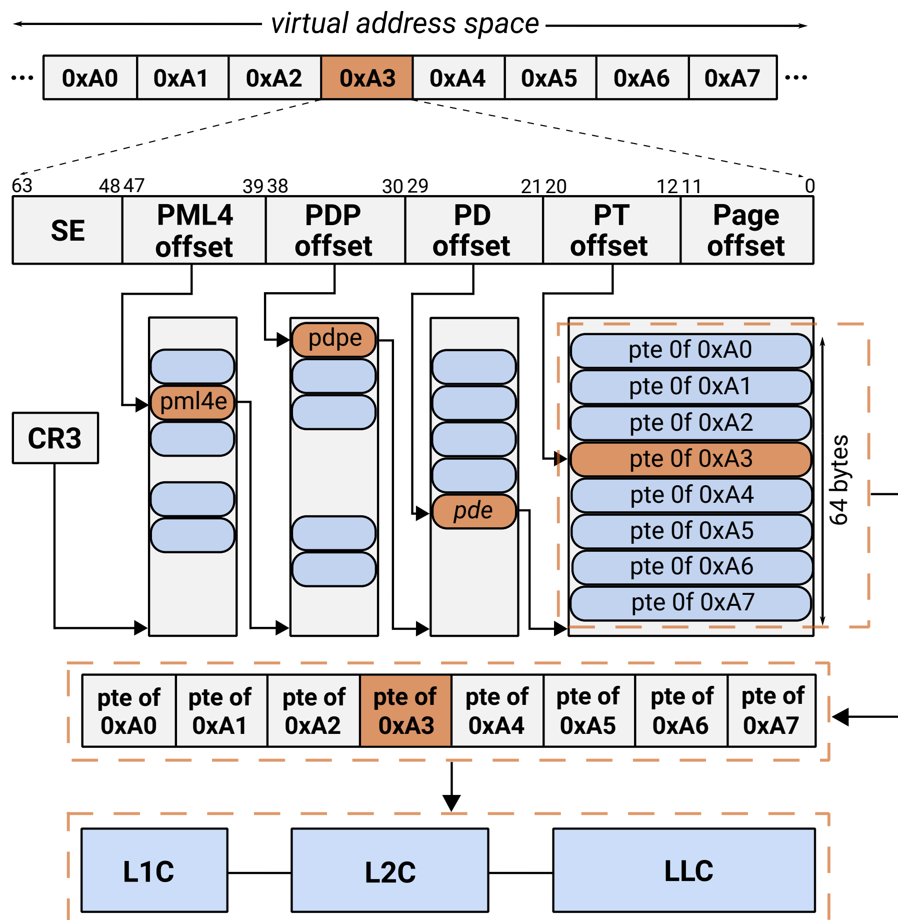


Figure 3.1: Locality in the last level of the radix tree page table in x86-64 architectures, assuming a 4-level radix tree page table and a TLB miss for virtual page 0xA3. This thesis interchangeably uses the terms *PTE locality* and *page table locality* to refer to this locality.

that these neighboring PTEs are contiguous in both virtual and physical address spaces, but they may point to non-contiguous physical pages (depending on the system state, fragmentation). Figure 3.1 demonstrates how a page walk is performed in x86-64 architectures, similar to Figure 2.12 in Section 2.3.3.1, and illustrates the locality of the PTEs in the last level of the radix tree page table. For the rest of this chapter, we interchangeably use *PTE locality* and *page table locality* to refer to the locality in the last level of the radix tree page table. Finally, the term *free TLB prefetching* (or simply *free prefetching*) refers to the process of exploiting page table locality by fetching the 'free' PTEs that reside within a 64-byte cache line due to a page walk into a storage medium (it can be either the TLB or a dedicated buffer placed alongside the TLB hierarchy).

## 3.3 Motivation

This section motivates the need for new TLB prefetching approaches since there was a paucity of research in the TLB prefetching domain for more than 10 years. In addition, it highlights the potential performance improvements when page table locality, presented in Section 3.2.1, is exploited by previously proposed TLB prefetchers. To do so, we implement and evaluate the previously proposed TLB prefetchers SP, ASP, and DP, presented in Section 2.4.1.2, and we set their configuration parameters as proposed in the original papers; Section 3.7.2 presents in detail their configuration. Our motivational analysis does not consider the MP prefetcher, described in Section 2.4.1.2, since MP is evaluated in Section 3.8.3 to compare our proposals with another scheme that opts to improve TLB performance. Finally, all evaluated TLB prefetchers store the prefetched PTEs into a 64-entry PB that uses the FIFO replacement policy, similar to prior work [165]; we evaluate the impact of different PB sizes in Section 3.8.1.1.

Apart from the scenario that considers the original implementation of the SP, DP, and ASP with a 64-entry PB, we evaluate additional scenarios to demonstrate the potential of TLB prefetching and quantify how much benefit can PTE locality provide on TLB prefetching. The evaluated scenarios are described below.

- We consider the case where no TLB prefetcher performs in the system but PTE locality is exploited by storing the available PTEs into the PB on demand page walks. This scenario quantifies the potential of leveraging PTE locality in a standalone mode, *i.e.*, when there is no TLB prefetcher operating in the system and PTE locality is exploited only for demand page walks.

- We enhance all considered TLB prefetchers (SP, DP, ASP) with an unbounded PB to store all the available PTEs in the cache line returned at the end of each page walk for both demand and prefetch page walks. This scenario quantifies how much PTE locality exploitation can improve the performance of previously proposed TLB prefetchers.

- To quantify the upper bound of the performance that an oracle TLB prefetcher could offer, we consider an idealized scenario; a *Perfect TLB* where all the accesses are hits, *i.e.*, the system does not experience any TLB miss, thus there is no page walk happening in the system.

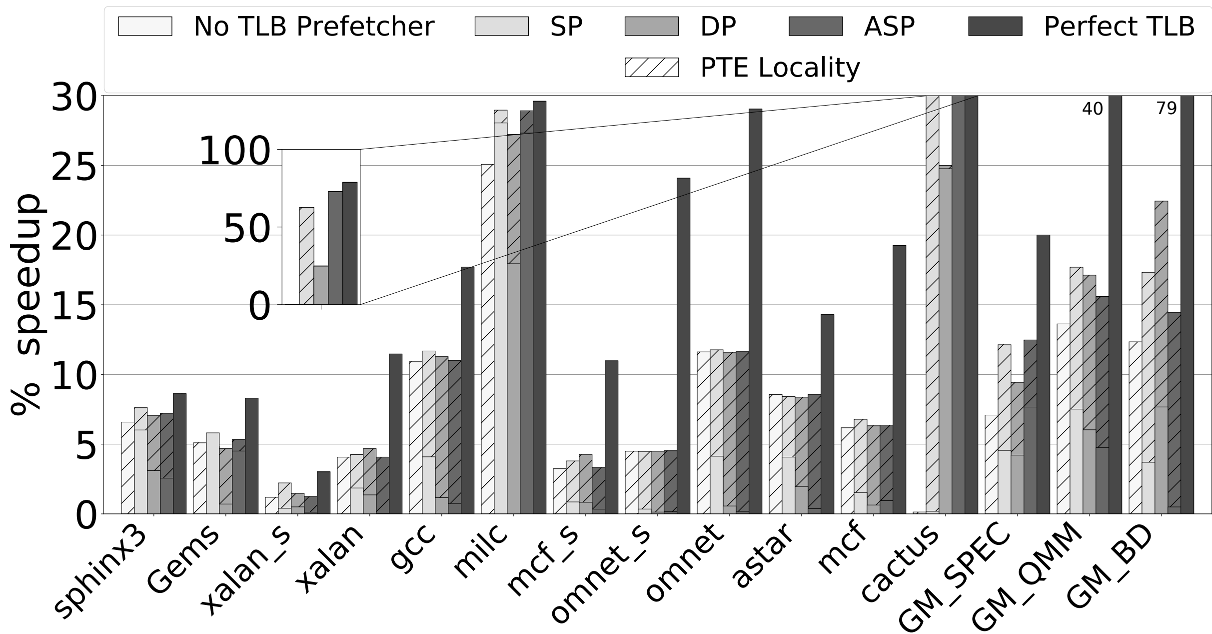


Figure 3.2: Performance of SP, ASP, DP, and Perfect TLB with and without exploiting PTE locality, illustrated in Figure 3.1. Higher is better.

The above explained scenarios are evaluated using the ChampSim simulator [13, 124], considering a large set of workloads. Specifically, we use industrial workloads provided by Qualcomm for the 1st Contest on Value Prediction (CVP-1) [19], the SPEC CPU 2006 [136] and SPEC CPU 2017 [42] benchmark suites, the GAP [74] benchmark suite, and the XSBench [45]. For the rest of this chapter, we use QMM, SPEC, and Big Data (BD) to refer to the Qualcomm workloads from CVP-1, the SPEC CPU 2006 and SPEC CPU 2017 workloads, and the GAP plus XSBench workloads, respectively. Section 3.7 explains in detail our simulation infrastructure, our experimental setup, and the considered workloads. Figures 3.2 and 3.3 present the evaluation results of the previously presented scenarios.

Figure 3.2 shows the performance delivered by the previously proposed TLB prefetchers (SP, DP, ASP), the Perfect TLB, and the scenario without TLB prefetcher (NoPref). In addition, Figure 3.2 quantifies the impact of PTE locality on the performance results by combining all schemes with the scenario that uses an unbounded PB to exploit PTE locality. Note that the speedup of the NoPref scenario is non-zero only when PTE locality is exploited. All speedups are computed over a baseline that does not use TLB prefetching. Figure 3.2 reports speedups only for the considered SPEC benchmarks coupled with a geo-

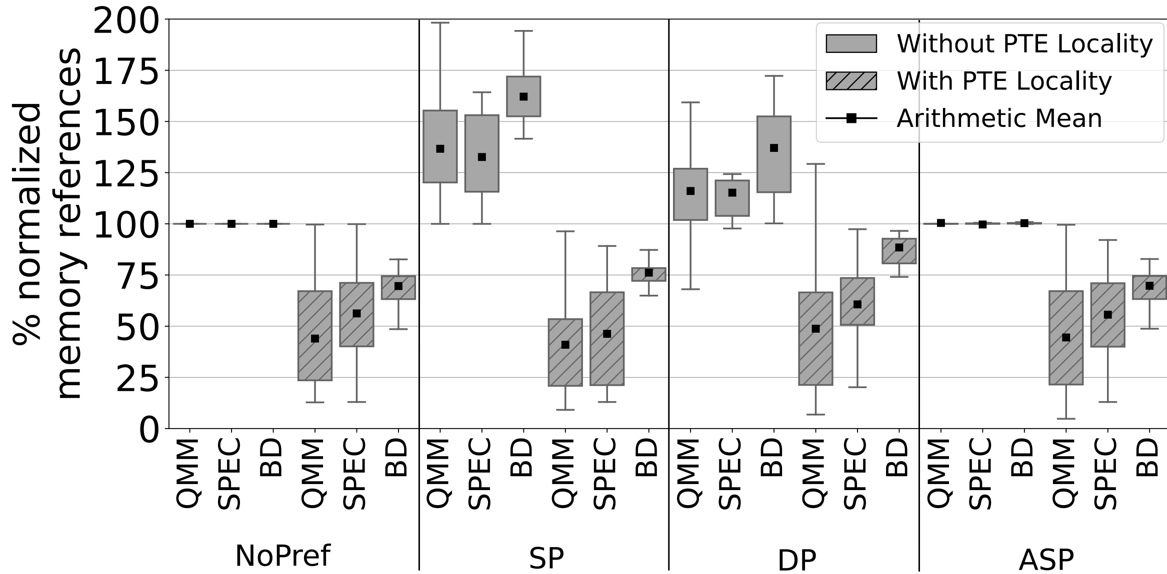


Figure 3.3: Distribution of the normalized number of memory references due to page walks (demand and prefetch) depicted with box plots. Lower is better.

metric mean across all SPEC, QMM, and BD workloads for readability since our workload set encloses 12 SPEC CPU workloads, 125 QMM workloads, and 13 BD workloads. Section 3.8 presents experimental results across all considered workloads.

Figure 3.3 quantifies the impact of TLB prefetching on the memory footprint of page walks. To do so, it presents the distribution of the normalized number of memory references caused by page walks (demand plus prefetch) for SP, DP, ASP, and the scenario without TLB prefetcher (NoPref) with and without exploiting PTE locality. The Perfect TLB scenario is omitted from Figure 3.3 since all TLB accesses are hits, thus there is no page walks happening. The term *memory reference* refers to a page walk reference that is served by the memory hierarchy (L1, L2, LLC, DRAM), as presented in Section 2.4.1.1, since our methodology (Section 3.7) takes into account cache locality in page walks. Note that a page walk memory reference is triggered for accesses that miss in the MMU-Caches, as described in Section 2.3.3.4. All scenarios are evaluated with and without exploiting PTE locality to highlight its impact on page walk memory references. The normalization factor, 100% in Figure 3.3, is the total number of memory references without TLB prefetching.

Subsequent sections elaborate on the results presented in Figures 3.2 and 3.3 and draw the key findings of this motivational analysis which constitute the main source of inspiration for our proposals, presented in Sections 3.4 and 3.5.

### 3.3.1 Quantifying the Potential of TLB Prefetching

Looking at Figure 3.2 it can be observed that the Perfect TLB scenario significantly outperforms the original implementations of all considered TLB prefetchers across all benchmark suites. Specifically, the Perfect TLB scenario yields a geometric mean speedup of 20.0%, 40.0%, and 79.0% for the SPEC, QMM, and BD workloads, respectively. The geometric speedups of the Perfect TLB scenario are 12.4%, 32.5%, and 71.4% higher than the speedups provided by the best performing prior TLB prefetcher (it is different per benchmark suite) for the SPEC, QMM, and BD workloads, respectively, revealing that there is a large room for improving TLB performance via hardware prefetching. The potential for performance improvement is higher for the QMM and BD workloads than the SPEC workloads since the latter have smaller data working sets sizes than the former, thus placing less pressure on the TLB hierarchy. However, the original implementations of the state-of-the-art TLB prefetchers (SP, DP, ASP) fall short at identifying the TLB miss patterns since they provide low performance gains compared to Perfect TLB, as shown in Figure 3.2.

***Finding 1.*** *Intelligent TLB prefetching has the potential to provide great performance gains.*

### 3.3.2 Does Any Prior TLB Prefetcher Dominate?

To answer whether a single TLB prefetcher performs best across all workloads, we focus on the speedup results of the original SP, DP, and ASP implementations without exploiting PTE locality, presented in Figure 3.2. Overall, the original implementations of SP, DP, and ASP provide a geometric mean speedup of 4.5%, 4.2%, and 7.6% for the SPEC workloads, 7.5%, 6.1%, and 4.8% for the QMM workloads, and 3.7%, 7.6%, and 0.5% for the BD workloads, respectively. Looking at the geometric mean speedups, we observe that different TLB prefetchers perform best across different benchmark suites. Specifically, ASP, SP, and DP provide the highest performance gains for the SPEC, QMM, and BD workloads, respectively. Despite ASP providing the overall highest geometric mean speedups for the SPEC workloads, it can be observed that for some SPEC workloads SP and/or DP outperform ASP. For example, SP provides the highest speedup for the `sphinx3` benchmark among the considered TLB prefetchers. The main takeaway is that different TLB prefetchers perform best for different workloads due to the variety of memory access patterns present in different

workloads types. For benchmarks showing irregularly distributed stride TLB miss patterns (*e.g.,* `cactus`), ASP and DP outperform SP. By contrast, for benchmarks with sequential TLB miss patterns (*e.g.,* `sphinx3`), SP outperforms both ASP and DP due to conflicts in their prediction tables. These conflicts force ASP and DP to discard the captured stride patterns and, once the execution shows again a regular pattern, they require several TLB misses to identify again the corresponding strides. Finally, SP, ASP, and DP are incapable of capturing highly irregular TLB miss patterns (*e.g.,* `mcf`). Although SP, ASP, and DP are ineffective in these irregular scenarios, they keep on triggering prefetch page walks to serve inaccurate prefetches.

***Finding 2.*** *There exists no single TLB prefetcher that performs best across all workloads.*

### 3.3.3 What is the Impact of TLB Prefetching on Page Walk References?

This section quantifies the impact of previously proposed TLB prefetchers on page walk memory references. Accurate TLB prefetches save long-latency demand page walks that would otherwise be on the critical path of accessing memory at the cost of introducing prefetch page walks that are performed in the background, without enlarging the critical path of memory accesses. However, inaccurate TLB prefetching introduces additional references to the memory hierarchy due to prefetch page walks without reducing the number of demand page walks. Looking at Figure 3.3, we observe that without exploiting PTE locality SP, DP, and ASP trigger 63%, 36%, and 1% additional memory references over the baseline that does not apply TLB prefetching, respectively, for the BD workloads. Similar behavior is observed for the SPEC and QMM workloads. SP and DP introduce a large number of page walk references to the memory hierarchy since they keep on issuing prefetches that trigger prefetch page walks, even when they fail at capturing the TLB miss patterns. Contrarily, ASP has a negligible impact on the page walk memory references due to the state field of its prediction table which acts like a throttling mechanism, ensuring that mostly accurate prefetches will be issued.

***Finding 3.*** *TLB prefetching triggers prefetch page walks that induce additional references to the memory hierarchy, exacerbating the Memory Wall bottleneck.*

### 3.3.4 What is the Potential of Exploiting Page Table Locality?

Intuitively, page table locality has the potential to significantly benefit hardware TLB prefetching due to its intrinsic properties. First, the neighboring PTEs that are transferred together with the requested PTE could be stored into the PB and potentially save forthcoming TLB misses. In addition, these neighboring PTEs are prefetched for 'free' since they do not require any prefetch page walk to be stored into the PB, thus PTE locality exploitation decreases the number of prefetch page walks introduced by TLB prefetching.

To quantitatively answer whether PTE locality has the potential to improve TLB prefetching performance, we focus on Figure 3.2 and compare the performance of the considered TLB prefetchers with and without exploiting PTE locality. We observe that all prior TLB prefetchers experience great performance gains when they exploit PTE locality. Specifically, SP, DP, and ASP yield geometric mean speedups of 12.1%, 9.4%, and 12.5% for the SPEC workloads, 17.7%, 17.1%, and 13.6% for the QMM workloads, and 17.3%, 22.4%, and 14.4% for the BD workloads, respectively. Therefore, when state-of-the-art TLB prefetchers exploit PTE locality, they provide higher performance than their original versions that do not exploit PTE locality. For example, DP with PTE locality outperforms DP original by 14.8% for the BD workloads. In addition, we observe that even the scenario without TLB prefetcher (NoPref) that exploits PTE locality only on demand page walks provides significant performance gains. In practice, the NoPref scenario coupled with PTE locality delivers a geometric mean speedup of 7.1%, 13.6%, and 12.3% for the SPEC, QMM, and BD workloads, respectively. Greater performance is reported when TLB prefetchers exploit PTE locality because the NoPref scenario leverages PTE locality only on demand page walks, while TLB prefetchers also issue prefetch page walks, thus they further exploit PTE locality on prefetch page walks.

***Finding 4.*** *Exploiting PTE locality for TLB prefetching purposes has the potential to significantly improve performance.*

Figure 3.3 examines the impact of PTE locality exploitation on the page walk memory references. We observe that, when PTE locality is exploited, the number of memory references due to page walks is massively reduced for all prior TLB prefetchers compared to their versions that do not exploit PTE locality. SP achieves a higher reduction in page walk memory references than DP and ASP because it issues prefetch requests using the +1 stride,

which are likely to already be fetched in the PB due to PTE locality exploitation. DP and ASP use larger strides, slightly reducing the impact of PTE locality on them. Finally, the reduction of page walk memory references is the reason why all considered TLB prefetchers experience higher performance gains when leveraging PTE locality, as shown in Figure 3.2.

***Finding 5.*** *Exploiting PTE locality significantly reduces the page walk memory references.*

### 3.3.5 Putting Everything Together

Our analysis indicates that smart TLB prefetching coupled with PTE locality exploitation is a promising solution to the address translation bottleneck. However, the reported improvements of exploiting PTE locality for TLB prefetching assume an ideal and indefinitely large PB. Moreover, TLB prefetching is limited by the PB size due to latency and area overheads. Therefore, combining PTE locality exploitation with TLB prefetching in the context of a properly sized PB requires a smart method to select the most useful PTEs per page walk.

## 3.4 Sampling-Based Free TLB Prefetching (SBFP)

This section presents *Sampling-Based Free TLB Prefetching (SBFP)*, an agile scheme that exploits PTE locality, presented in Section 3.2.1, to reduce the cost and improve the effectiveness of TLB prefetching. SBFP uses sampling to predict which of the cache-line adjacent PTEs, that can be prefetched for 'free', are more likely to prevent future TLB misses and fetches them into the TLB PB. We demonstrate that SBFP reduces the negative impact of prefetch page walks, it can be combined with any TLB prefetcher, and it can operate on both demand and prefetch page walks.

### 3.4.1 Pushing the Envelope on Free TLB Prefetching

As described in Section 3.2.1, 7 PTEs that are 'free', *i.e.*, they do not require any additional memory operations to be prefetched, are stored in the cache hierarchy at the end of each page walk. The naive approach is to prefetch all available free PTEs into the TLB PB. However, TLB prefetching is limited by the size of the PB, the cost of PB lookups, and the PB area overhead. Therefore, naively storing all available free prefetches, *i.e.*, the cache line

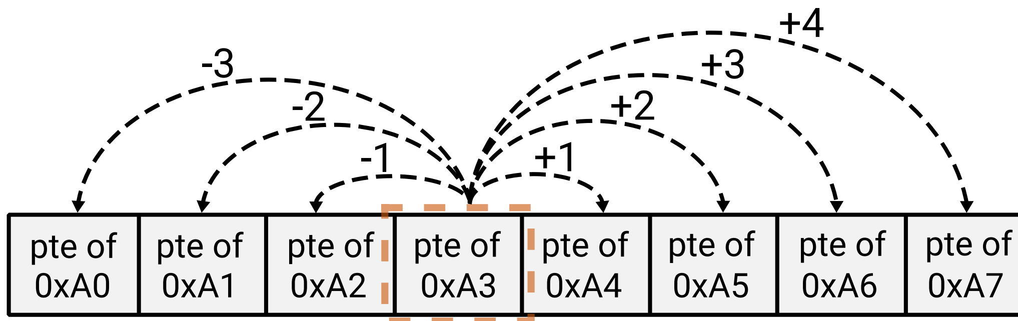


Figure 3.4: Illustration of the *free distance* concept, assuming a page walk for virtual page 0xA3.

adjacent PTEs, per page walk into the PB provides suboptimal performance benefits due to evicting useful prefetches from the PB while polluting the PB with inaccurate prefetches (evaluated in Section 3.8.3). Therefore, to exploit the benefits of PTE locality in the context of a realistic PB size, a scheme that dynamically identifies and prefetches only the useful free PTEs per page walk is required. To address this need, we design and propose the *Sampling-Based Free TLB Prefetching (SBFP)* mechanism, a dynamic scheme that predicts via sampling the usefulness of the free PTEs per page walk and fetches in the PB only the most likely ones to save future TLB misses.

### 3.4.2 SBFP Design and Operation

The operation of SBFP relies on the *free distance* concept. We define free distance as the distance, within the cache line, between the PTE that holds the demand translation and another PTE that can be obtained for free. Depending on the position of the requested PTE in the cache line, there are 14 possible free distances: from -7 to +7, excluding 0. Figure 3.4 illustrates the free distance concept; it assumes a page walk for virtual page 0xA3 and shows the free distances of the PTEs residing in the same cache line with the requested PTE. For example, the free distance of the PTE of virtual page 0xA1 is -2.

#### 3.4.2.1 Design Overview

The SBFP scheme associates each free PTE within a cache line with a free distance and leverages this information to predict the usefulness of the corresponding PTEs. To do so, SBFP uses three data structures: the *Sampler*, the *Free Distance Table (FDT)*, and the *Prefetch Buffer (PB)*. Figure 3.5 presents the components and the functionality of the SBFP module.

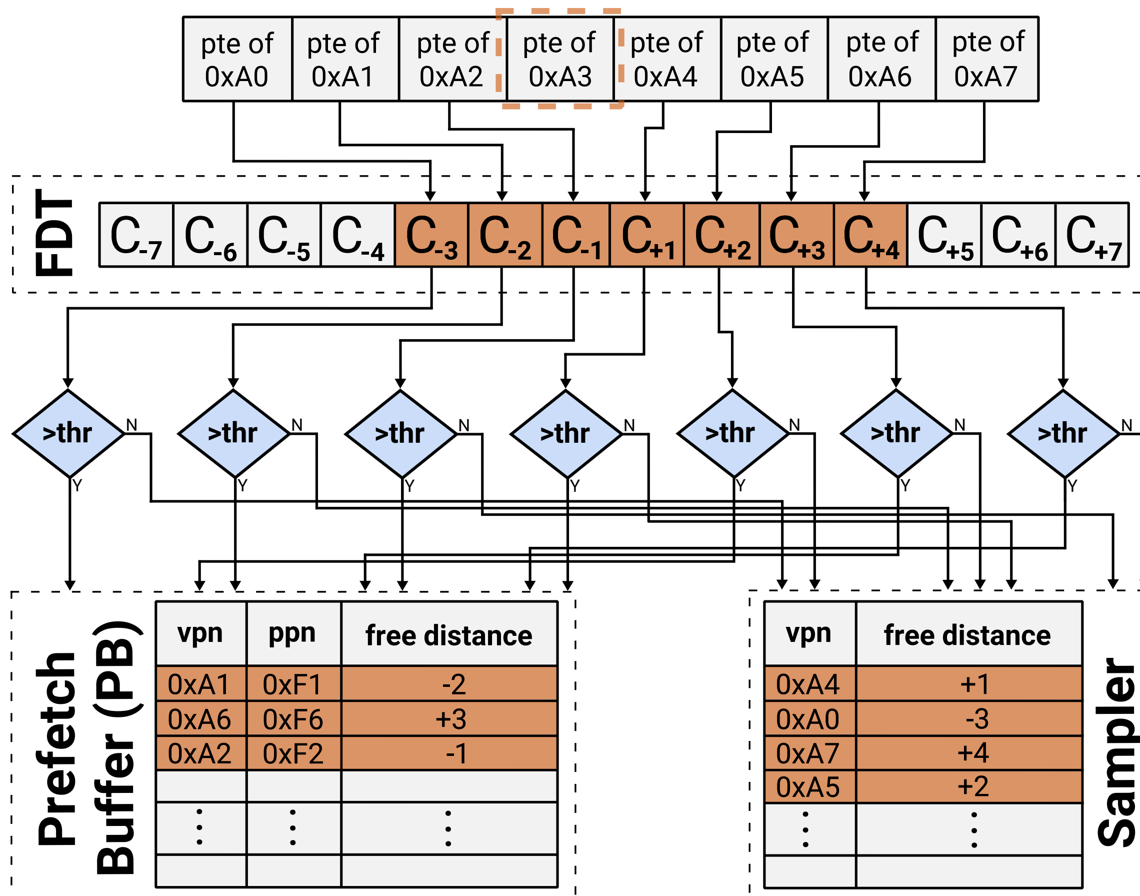


Figure 3.5: Sampling-Based free TLB prefetching (SBFP) mechanism. The example considers a TLB miss for virtual page 0xA3, similar to Figure 3.4. The terms *vpn*, *ppn*, and *thr* refer to virtual page number, physical page number, and threshold, respectively. Diamonds indicate decision points.

The Sampler is a small buffer that is responsible for detecting phases when free distances, which were previously useless, can provide useful prefetches. To do so, each Sampler entry stores the virtual page number and its corresponding free distance for every free PTE that is decided not to be placed in the PB. The decision whether to place a free PTE into the PB or the Sampler is made by the FDT, a table composed of 14 saturating counters, one per possible free distance. Each FDT counter monitors the hit ratio of one free distance. Finally, the PB is a fully associative buffer that stores the virtual page number, the physical page number, and the corresponding free distance of the free PTEs, similar to the PB used by TLB prefetchers, introduced in Section 2.4.1.1.

#### 3.4.2.2 SBFP Operation

To explain the operation of SBFP, we consider the example presented in Figure 3.5 that assumes a page walk triggered for virtual page 0xA3. At the end of the corresponding page walk, we identify the position of the requested PTE inside the cache line by extracting the 3 least significant bits of the virtual page. Therefore, the PTE of 0xA3 resides in position 4 within the cache line (counting starts from zero). Then we calculate the free distances of all free PTEs residing in the same cache line and we associate each PTE with a free distance. For example, the PTEs of virtual pages 0xA2 and 0xA5 are associated with free distances -1 and +2, respectively.

The next step is to determine whether a free PTE has to be stored in the PB or the Sampler. To do so, we compare the FDT saturating counter corresponding to its free distance with a threshold (*thr* in Figure 3.5). If the counter exceeds the threshold, the free prefetch is stored in the PB; otherwise, it is placed in the Sampler. For example, the PTE of 0xA2 has free distance -1, thus we compare the saturating counter of the FDT that corresponds to free distance -1 ($C_{-1}$) with a threshold to determine if the PTE of 0xA2 should be inserted in the PB or in the Sampler. The same procedure is followed for each free PTE in the cache line. Since PTEs only contain physical addresses, their virtual page numbers must be computed before inserting them in the PB or the Sampler; the virtual page numbers of the free prefetches are computed as the virtual page number of the demand translation plus their corresponding free distance. For example, the virtual page number of the PTE with free distance -1 is: 0xA3 (demand PTE) - 1 = 0xA2.

When a PB or Sampler hit occurs, the FDT counter that corresponds to the free distance of the hit entry is incremented by one. For instance, in the case of PB or Sampler hit caused by a prefetch that was associated with free distance -5, we simply increment by one the FDT counter $C_{-5}$. To prevent permanent saturation, we use a decay scheme that shifts right one bit all the FDT counters when one of the FDT counters saturates.

To summarize, the proposed Sampling-Based Free TLB Prefetching (SBFP) scheme adjusts the values of FDT counters depending on which free distances are frequently producing PB and Sampler hits, which makes SBFP capable of agilely predicting the most useful free PTEs per page walk. Note that the Sampler is searched only upon PB misses, so its lookup is not placed in the critical path of accessing memory.

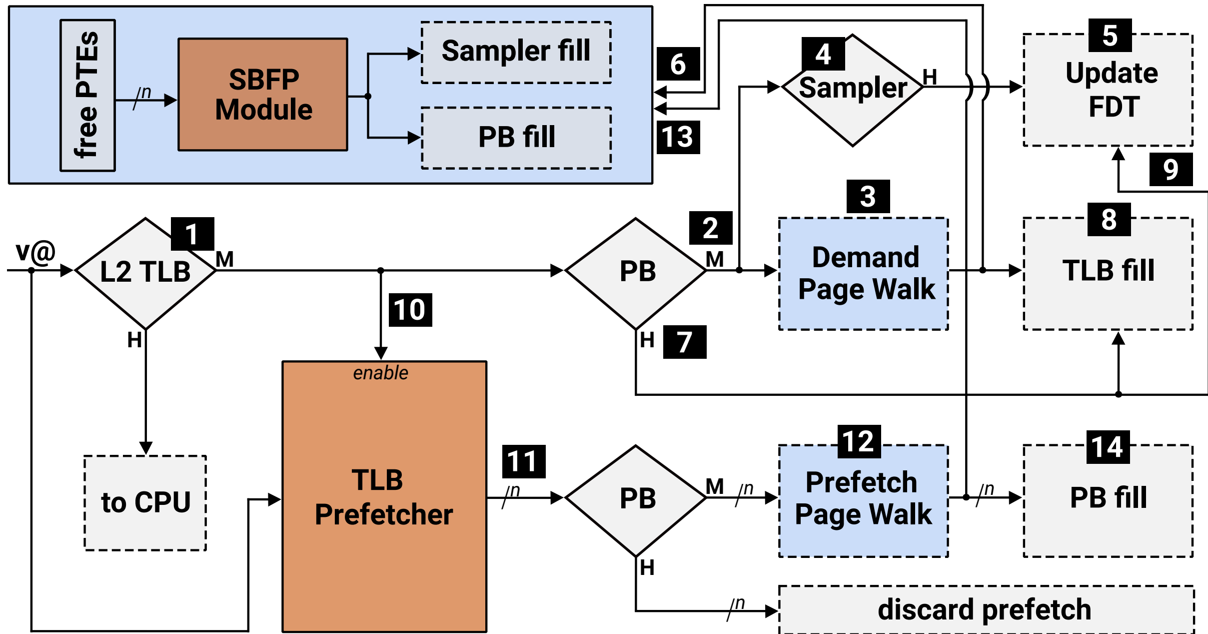


Figure 3.6: Operation when SBFP is combined with a generic TLB prefetcher. Diamonds indicate decision points while dotted lines indicate actions.

### 3.4.3 Combining SBFP with TLB Prefetching Schemes

This section demonstrates that SBFP can be combined with any TLB prefetcher to exploit the benefits of PTE locality on both demand and prefetch page walks. To do so, it considers a system that uses a generic TLB prefetcher as well as SBFP. Figure 3.6 shows in steps the operation of this system, pointing out the interaction between SBFP and the TLB prefetcher. Stress that the TLB prefetcher and SBFP use a shared PB to store the prefetched PTEs.

On a TLB miss 1, the requested translation is looked up in the PB. On PB misses 2, a demand page walk is initiated to fetch the translation from the page table 3. In the background, the Sampler is looked up for possible hits 4. On Sampler hits, we increment the FDT counter that corresponds to the free distance of the hit entry 5. When the demand page walk finishes, the SBFP scheme operates and decides which free PTEs should be placed in the PB and the Sampler 6. On PB hits 7, the demand page walk is avoided, the translation is transferred to TLB 8, and if the hit was produced by a free prefetch the FDT counter that corresponds to the free distance of the hit entry is incremented 9. In either case of PB hit or miss, the TLB prefetcher is activated 10 and produces new prefetch requests 11. Each prefetch that misses in the PB triggers a prefetch page walk to fetch

the corresponding translation from the page table 12. At the end of a prefetch page walk, the prefetched PTE is grouped with 7 PTEs that can be prefetched for free thanks to page table locality (Section 3.2.1). At this point, SBFP is again activated to decide which of the free prefetches should be placed in the PB or the Sampler, essentially applying lookahead prefetching with depth 2 13. Finally, the prefetched PTEs are stored in the PB 14.

To further elaborate on the operation of SBFP when combined with a generic TLB prefetcher, we consider the following example. The system experiences a TLB miss on virtual page 0xA3 which also misses in the PB. As a result, a demand page walk is initiated to fetch the corresponding PTE from the page table. When the demand page walk finishes, SBFP compares the FDT counters with a threshold to identify the most useful free distances for the current miss. Assuming that only free distance -1 exceeds the threshold, SBFP fetches the PTE of the virtual page 0xA2 (0xA3-1) in the PB while the virtual pages of the other free PTEs are stored in the Sampler. Next, we further assume that the TLB prefetcher issues a prefetch request for virtual page 0xB7. Similarly, SBFP places the PTE of 0xB6 (0xB7-1) in the PB since only free distance -1 exceeds the threshold; the other free PTEs within the cache line are stored in the Sampler.

The main takeaway of this section is that SBFP can be combined with any TLB prefetcher. Section 3.8 highlights that enhancing state-of-the-art TLB prefetchers with SBFP significantly improves their effectiveness. Finally, in Section 3.5 we design a TLB prefetcher aimed at maximizing the benefits of the proposed SBFP module.

### 3.4.4 Discussion

Modern workloads typically operate on multiple data structures that might favor different sets of free distances. Having perfect knowledge of the most useful free distances per data structure would require a separate FDT per each of these data structures. Alternatively, we propose a generalized SBFP that learns from any stream of accesses and uses a decay mechanism to ensure that only useful free distances will be used. We quantified the performance benefits of the scenario that uses a different FDT per PC that produces at least one TLB miss, and we observed modest performance gains over the generalized FDT. We observe such behavior because once the counters of the generalized FDT saturate, the decay mechanism lowers their values to increase their sensitivity to new data structures.

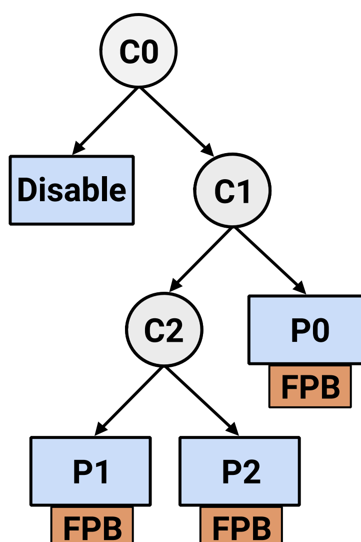


Figure 3.7: Design of the Agile TLB Prefetcher (ATP). P0, P1, and P2 represent generic TLB prefetchers. C0, C1, and C2 are saturating counters. Disable refers to disabling of TLB prefetching.

## 3.5 Agile TLB Prefetcher (ATP)

This section introduces *Agile TLB Prefetcher (ATP)*, a novel composite TLB prefetcher for data accesses, implemented as a decision tree. Unlike state-of-the-art TLB prefetchers, presented in Section 2.4.1.2, that correlate patterns with one feature (*e.g.,* constant strides, PC, distances between virtual pages that produce consecutive TLB misses), ATP captures patterns that correlate well with different features by combining three low-cost TLB prefetchers. To do so, ATP utilizes adaptive selection and throttling schemes to dynamically enable the most appropriate TLB prefetcher and disable TLB prefetching when it is not helpful.

### 3.5.1 Design Overview

Figure 3.7 presents the hardware components of ATP as well as its design which is inspired by the decision tree algorithm [20], similar to tournament branch predictors [192]. ATP consists of three easily implementable TLB prefetchers, named P0, P1, and P2 in Figure 3.7, to correlate the TLB miss patterns with multiple features, a single PB which is shared among its constituent prefetchers, and the dedicated selection and throttling schemes that require modest additional logic to be implemented: (i) a single saturating counter *C0* for the throttling mechanism, (ii) two saturating counters *C1*, *C2* that dynamically select the most

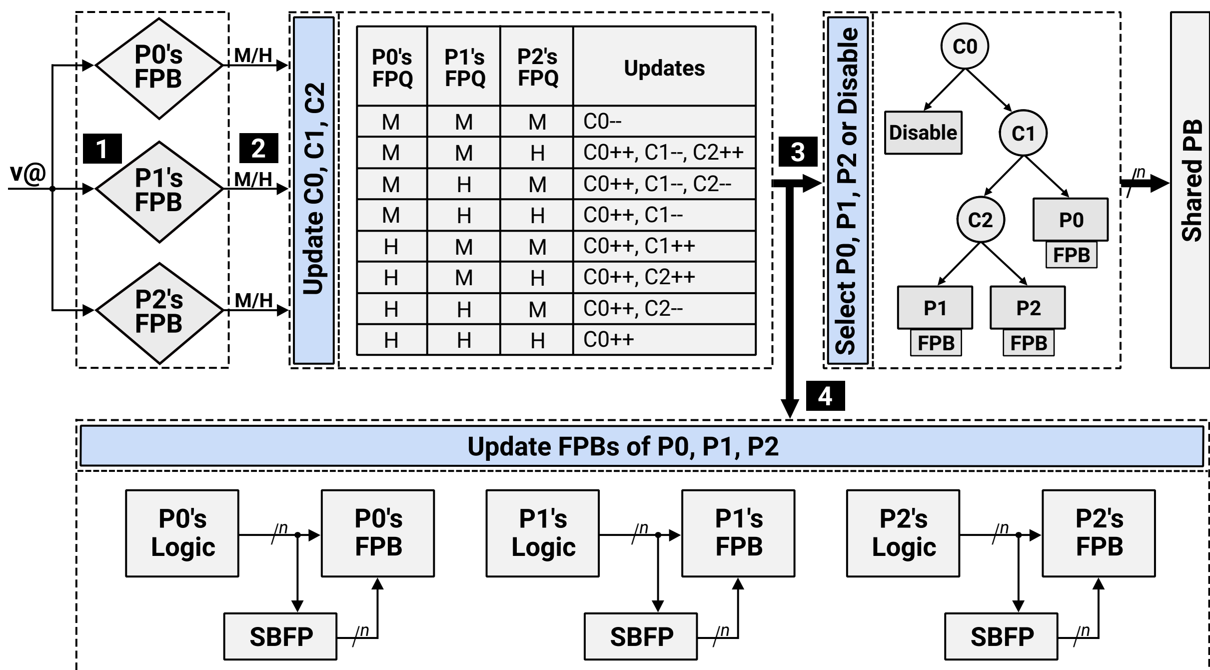


Figure 3.8: Functionality flowchart showing the operation of ATP. Diamonds indicate decision points.

accurate TLB prefetcher, and (iii) a *Fake Prefetch Buffer (FPB)* per constituent prefetcher, which monitor the accuracy of PO, P1, and P2 to update the values of C0, C1, and C2 accordingly. Each FPB holds only predicted virtual pages and not the corresponding address translations; hence, the term *fake*.

### 3.5.2 ATP Operation

Figure 3.8 shows step-by-step the operation of ATP. First, ATP looks for the translation of the missing virtual page in the FPBs of its constituent TLB prefetchers **1**. Depending on the search outcome, the saturating counters C0, C1, and C2 are updated in step **2**. For example, in case of a hit in at least one FPB, C0 increases its value for issuing new prefetches. Otherwise, C0 is decreased, *i.e.*, increases its confidence for disabling TLB prefetching.

Next, ATP uses the updated values of the saturating counters to make a decision for the current miss. The C0 counter is responsible for choosing whether to enable or disable TLB prefetching for the current access. If the most significant bit of C0 is one, the decision is to issue new prefetch requests. In this case, C1 is probed to select the individual TLB prefetcher that will generate prefetches for the current TLB miss **3**. If the most significant

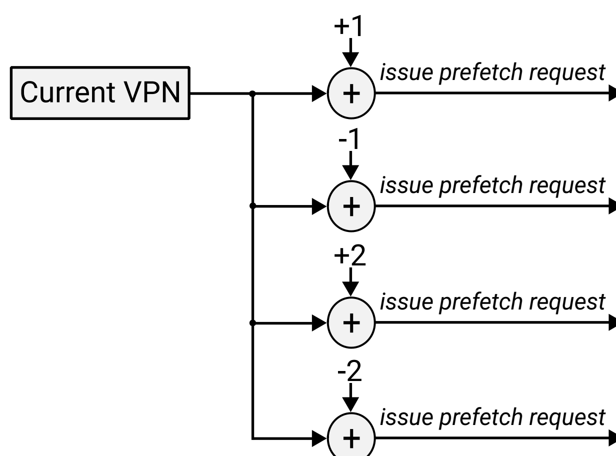


Figure 3.9: Operation of STP.

bit of C1 is one, the TLB prefetcher residing in the right leaf (P0) is selected; otherwise, C2 is responsible for selecting which prefetcher should be enabled. Likewise, if the most significant bit of C2 is one, prefetcher P2 is selected; otherwise, prefetcher P1 issues prefetches for the current TLB miss. Finally, if the most significant bit of C0 is zero no prefetch request will be issued for the current access.

Subsequently, ATP updates the content of all FPBs 4. All the constituent prefetchers of ATP store in their own FPB the virtual pages corresponding to the prefetches that they would issue if they were allowed to individually produce prefetches plus the free prefetches that SBFP would select after the completion of each fake page walk. Using these 'fake prefetches' we track the usefulness of each TLB prefetcher for future TLB misses and update the values of the saturating counters C0, C1, and C2 accordingly.

### 3.5.3 Building Blocks of ATP

ATP is composed of three easily implementable TLB prefetchers that are presented below.

#### Stride Prefetcher (STP)

STP is a more aggressive version of SP, presented in Section 2.4.1.2. STP uses the strides $\{-2, -1, +1, +2\}$ for producing prefetch requests. Specifically, upon a TLB miss for virtual page *A*, STP will prefetch the PTEs of the pages {A–2,A–1,A+1,A+2}. Figure 3.9 illustrates the operation of STP.

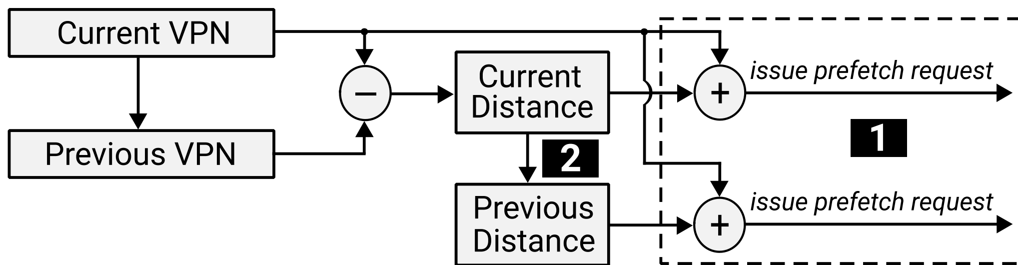


Figure 3.10: H2P's operation.

### H2 Prefetcher (H2P)

H2P is a low-cost version of DP that correlates patterns with the distances between virtual pages that cause consecutive TLB misses. In practice, H2P keeps track of the last two observed distances between virtual pages that caused a TLB miss. Assuming that $d(X,Y)$ represents the signed distance between the pages $X$ and $Y$, and 0xA, 0xB, and 0xE are the last three virtual pages that caused a TLB miss, H2P will prefetch the PTEs of the virtual pages: 0xE + $d(0xE,0xB)$ and 0xE + $d(0xB,0xA)$. Finally, H2P will store the current distance in the register accommodating the previous distance. The above explained operation is shown step-by-step in Figure 3.10.

### Modified Arbitrary Stride Prefetcher (MASP)

MASP is an aggressive evolution of ASP, presented in Section 2.4.1.2, that leverages a prediction table indexed with the PC and targets varying stride patterns. To issue prefetch requests, ASP requires two consecutive hits in a certain prediction table entry to display the same stride. While this policy increases the accuracy of ASP, it misses prefetching opportunities. Therefore, we design MASP, implementing two modifications to ASP: (i) the requirement of observing the same stride at least twice consecutively is removed, and (ii) a second prefetch takes place for each TLB miss, using the virtual page that caused the current TLB miss and the stride residing in the hit entry. Each entry of the prediction table of MASP has three fields: the PC for indexing, the previous virtual page that caused a TLB miss while being accessed by that PC, and the corresponding stride, similar to ASP.

The operation of MASP is straightforward and is illustrated in Figure 3.11 coupled with an example. On a TLB miss, MASP looks up the prediction table for possible hits **1**. On a prediction table miss, the PC is stored in the first field of a prediction table entry and the

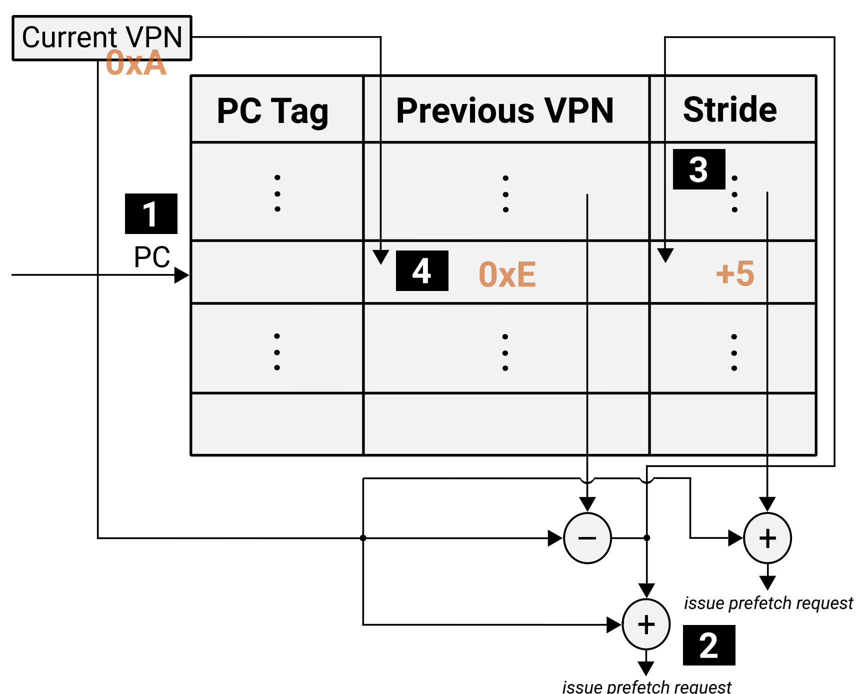


Figure 3.11: MASP's operation.

stride field is invalided. On a table hit, MASP issues two prefetch requests; the first uses the stride of the hit entry and the second the distance between the virtual pages that produced the current and the previous TLB misses while accessed by this PC 2. Next, MASP updates the stride using the current and previous missing virtual pages 3. Finally, in case of either prediction table hit or miss, the current virtual page number is stored in the second field of the corresponding entry 4.

Figure 3.11 elaborates on the prefetching algorithm of MASP by presenting a concrete example where there is a TLB miss for virtual page 0xA that hits in the prediction table of MASP. The respective entry has in the second field the virtual page $E$ and in the third field the stride +5. MASP will generate prefetches for the PTEs of virtual pages $0xA + 5$ and $0xA + d(0xA, 0xE)$, where $d(X, Y)$ computes the signed distance between pages $X$ and $Y$.

### Insights on ATP's Operation

ATP enables STP, H2P, and MASP when the TLB miss stream correlates well with small strides, the distance between virtual pages that produce TLB misses, and the PC, respectively. The aggressiveness of STP and H2P may negatively impact both performance and the

number of triggered page walks, as we demonstrate in Section 3.8.1. ATP minimizes these negative effects by enabling STP and H2P only when they are likely to produce accurate prefetches. When the TLB miss stream exhibits irregular patterns, the throttling scheme of ATP disables TLB prefetching until it observes again patterns that are predictable by at least one of its constituent prefetchers, leveraging the operation of the FPBs.

#### Placement of STP, H2P, and MASP on ATP's Nodes

The placement of STP, H2P, and MASP on the leaf nodes P0, P1, and P2, presented in Figures 3.7 and 3.8, plays a critical role for ATP's performance. This placement depends on the number of bits used for the C0, C1, and C2 counters. We empirically found that 8-bit, 6-bit, and 2-bit counters are good design points for C0, C1, and C2, respectively. For this configuration, leaf nodes P0, P1, and P2 are assigned to H2P, MASP, and STP, respectively. The rationale behind this placement is simple and relies on two properties: (i) when multiple prefetchers experience hits in their FPBs, activate the less aggressive one, and (ii) make sure that the aggressive prefetchers are activated when there is high confidence that they produce accurate prefetches. Counter C1 is 6 bits, thus we assign H2P to node P0 to make sure that H2P will be enabled only when it is proved that it produces useful prefetches since the right node is selected when the most significant bit of the corresponding counter is 1. In practice, this happens only when there are 16 ($2^6/2$) consecutive hits in the FPB of H2P with concurrent misses in the FPBs of MASP and STP, as shown in Figure 3.8 which depicts the updates of the counters depending on the FPB lookup outcomes. In a similar spirit, STP is placed on the right node of counter C2; on the left path, there is MASP which captures patterns that correlate well with the PC. However, counter C2 is 2 bits to enable fast changes between MASP and STP. Still, when both MASP and STP experience hits in the FPBs, MASP is selected since it resides on the left (default) node of counter C2.

### 3.5.4 Discussion

ATP could be extended with additional nodes to accommodate new TLB prefetching schemes. This would require more counters for the selection logic and potentially different placement of the individual prefetchers on the leaf nodes of ATP. In addition, ATP's design is transparent to which TLB prefetcher is used, thus an architect could replace any of the TLB prefetchers residing in ATP's leaf nodes with other more sophisticated TLB prefetchers. Finally, ATP's ag-

ile design that enables efficient and accurate switching between different prefetchers could be utilized in other domains that seek adaptive switching between policies. Examples are cache prefetching, branch prediction [198], and cache replacement policies among others.

## 3.6 ATP+SBFP : Additional Considerations

### Impact on Page Replacement Policy

TLB prefetching is a speculation technique. Therefore, TLB prefetches should ideally not influence the access bits of the prefetched pages. However, the memory consistency model of x86 architectures dictates that TLBs are allowed to accommodate address translation entries that have their status bits on, *i.e.*, all TLB prefetches are obliged to set the access bits of the corresponding pages. As a consequence, inaccurate TLB prefetches can negatively affect the page replacement policy (Section 2.3.3.5) and lead to suboptimal decisions. Prior work on TLB prefetching does not consider the impact on the page replacement policy due to the growing memory capacities. However, with the advent of heterogeneous memories, the OS has to migrate data between fast and slow memories, so accurately setting the access bit is very important today. Section 3.8.4 shows that our proposal, ATP coupled with SBFP, has a negligible impact on the page replacement policy that can be fully eliminated by periodically issuing page walks that reset the access bits on inaccurate TLB prefetches.

### Multiple Page Sizes

Neither ATP nor SBFP requires any modifications to support multiple page sizes. Since the page size is known after address translation, ATP issues two prefetch requests per prefetch candidate assuming 4KB and 2MB pages and, when the page granularity is known, one of the prefetch page walks is discarded. This approach does not imply additional complexity since modern architectures support speculative page walks [215]. Alternatively, page size prediction could be used to issue a single prefetch request per candidate [212]. Regarding the SBFP scheme, it checks whether the free prefetches are valid translation entries before adding them in the PB or the Sampler (either valid PT entries or PD entries), even when PD entries that map 2MB pages are next to PD entries that point to PT entries. Finally, the PB is a fully associative structure, which avoids page size indexing implications [215].

#### Page Table Designs

Both ATP and SBFP are compatible with radix tree page table designs with any number of levels. Previous sections focus on 4-level radix tree page tables but our proposals are compatible with 5-level radix tree page tables [1], and may deliver more benefits because the extra page table level might increase the latency and energy cost of page walks. Finally, if a hashed page table design [118, 140, 259, 300] is used, both ATP and SBFP would operate the same since hashed page tables preserve page table locality.

#### TLB Prefetching Strategy

TLB prefetchers typically operate on TLB misses and store the prefetches into a TLB PB for the reasons outlined in Section 2.4.1.1. Nonetheless, ATP could be also activated on TLB hits and prefetch directly into the TLB.

#### Context Switches

ATP and SBFP leverage small structures that quickly warm up and are flushed upon context switches, so they do not need to be tagged with address space identifiers. Note there is no need for resetting the saturating counters of ATP and SBFP upon context switches.

## 3.7 Methodology

### 3.7.1 Simulation Infrastructure

We evaluate our proposals using ChampSim [13, 124], a detailed trace-based simulator that models a 4-wide out-of-order processor. We extend ChampSim to simulate the operation of a realistic page table walker used in x86-64 architectures, modeling the variant latency cost of page walks, the page walk references to memory hierarchy, and the cache locality in page walks, similar to prior work [188]. Specifically, we simulate a 4-level radix tree page table, a hardware page table walker, and a 3-level split PSC. The page table walker supports up to 4 TLB misses, similar to Skylake $\mu$architecture [48], while one page walk can be initiated per cycle. Regarding the cache hierarchy, we simulate 3 levels of caches, and hardware

| Component | Description |
|---|---|
| **ROB** | 256-entry |
| **L1 iTLB** | 64-entry, 4-way, 1cc, 4-entry MSHR, LRU |
| **L1 dTLB** | 64-entry, 4-way, 1cc, 4-entry MSHR, LRU |
| **L2 TLB** | 1536-entry, 12-way, 8cc, 4-entry MSHR, LRU, 1 page walk / cycle |
| **Page Structure Caches (PSCs)** | 3-level Split PSC, 2cc. PML4: 2-entry, fully assoc; PDP: 4-entry, fully assoc; PD: 32-entry, 4-way |
| **L1 iCache** | 32KB, 8-way, 1cc, 8-entry MSHR, LRU |
| **L1 dCache** | 32KB, 8-way, 4cc, 8-entry MSHR, LRU, next line prefetcher |
| **L2 Cache** | 256KB, 8-way, 8cc, 16-entry MSHR, LRU, ip stride prefetcher [60] |
| **LLC** | 2MB, 16-way, 20cc, 32-entry MSHR, LRU |
| **DRAM** | 4GB DDR4, tRP=tRCD=tCAS=11, 12.8 GB/s |
| **Branch Predictor** | gshare, 16384-entry history table |

Table 3.1: System simulation parameters.

prefetching is applied for the first two cache levels. Table 3.1 summarizes our baseline experimental setup. Note that the baseline system does not apply TLB prefetching and our evaluation primarily focuses on 4KB pages since they are still the norm in most virtual memory systems. Finally, we also quantify the impact of large pages on the performance of our proposals in Section 3.8.2.4.

The energy consumption measurements have been conducted using the CACTI 6.5 tool [180] with 22nm technology. Section 3.8.2.5 presents the impact of our proposals on the energy consumption of address translation.

### 3.7.2 Evaluated TLB Prefetchers

We implement and evaluate the previously proposed TLB prefetchers SP, DP, and ASP, described in Sections 2.4.1.2 and 3.2 as well as the proposed ATP prefetcher. Table 3.2 presents their configuration parameters. TLB prefetches are placed into a dedicated TLB buffer named Prefetch Buffer (PB); Table 3.2 presents its configuration.

| Prefetch Engine / Component | Description |
|---|---|
| **Sequential Prefetcher (SP)** | – |
| **Distance Prefetcher (DP)** | Prediction Table: 64-entry, 4-way, LRU policy |
| **Arbitrary Stride Prefetcher (ASP)** | Prediction Table: 64-entry, 4-way, LRU policy |
| **Stride Prefetcher (STP)** | – |
| **H2 Prefetcher (H2P)** | – |
| **Modified Arbitrary Stride Prefetcher (MASP)** | Prediction Table: 64-entry, 4-way, LRU policy |
| **Agile TLB Prefetcher (ATP)** | MASP's Prediction Table: 64-entry, 4-way, LRU policy,<br>Fake Prefetch Buffer (FPB): 16-entry, fully assoc, FIFO policy |
| **Sampling-Based Free TLB Prefetching (SBFP)** | FDT: 10-bit counters, Sampler: 64-entry, fully assoc, FIFO policy |
| **Prefetch Buffer (PB)** | (16-64)-entry, fully assoc, FIFO policy |

Table 3.2: Configuration parameters of the previously proposed TLB prefetchers (SP, DP, ASP), the Agile TLB Prefetcher (ATP), the constituent TLB prefetch engines of ATP (STP, H2P, MASP), the Sampling-Based Free TLB Prefetching (SBFP) module, and the Prefetch Buffer (PB) used for storing the prefetched PTEs. The parameters for ATP and SBFP have been empirically selected after thorough sensitivity analysis.

### 3.7.3 Policies Exploiting PTE Locality

To demonstrate the benefits of exploiting page table locality while highlighting the benefits of the proposed SBFP mechanism, we combine all considered TLB prefetchers (including the proposed ATP prefetcher and its constituent prefetchers) with different scenarios that exploit page table locality (including SBFP) and compare the results. Specifically, apart from the proposed SBFP scheme, we consider the following scenarios: (i) free prefetching is not exploited (NoFP), *i.e.*, free prefetches are not stored in the PB, (ii) all free prefetches are naively placed in the PB (NaiveFP), (iii) each TLB prefetcher uses its own optimal set of free distances based on a static offline exploration that identifies the most useful free distances per considered TLB prefetcher (StaticFP). To do so, we explore all possible sets of free distances and measure the performance of each TLB prefetcher. Note that there are $2^{14}$ different sets of free distances because there are 14 possible free distances. Table 3.3 presents the optimal, performance-wise, set of statically selected free distances for all considered TLB prefetchers. Finally, all scenarios that exploit PTE locality (including SBFP) share the same PB with the TLB prefetchers.

| TLB Prefetcher | Static Free Distances |
|---|---|
| **SP** | $\{+1, +3, +5, +7\}$ |
| **DP** | $\{-2, -1, +1, +2\}$ |
| **ASP** | $\{-1, +1, +2\}$ |
| **STP** | $\{+1, +2\}$ |
| **H2P** | $\{+1, +2, +7\}$ |
| **MASP** | $\{+1, +2\}$ |
| **ATP** | $\{+1, +2\}$ |

Table 3.3: Set of statically selected free distances for the considered TLB prefetchers.

| | web | road | twitter | kron | urand |
|---|---|---|---|---|---|
| **# Vertices (M)** | 50.6 | 23.9 | 61.6 | 134.2 | 134.2 |
| **# Edges (M)** | 1,949.4 | 58.3 | 1,468.4 | 2,111.6 | 2,147.4 |

Table 3.4: Input graphs for the GAP benchmark suite [74].

### 3.7.4 Workloads

Our evaluation uses an extensive set of workloads, spanning various benchmark suites. Specifically, we consider workloads provided by Qualcomm for the 1st Contest on Value Prediction (CVP-1) [19], all the benchmarks from SPEC CPU 2006 [136] and SPEC CPU 2017 [42] benchmark suites, and big memory footprint workloads included in the GAP [74] benchmark suite and XSBench [45]. The Qualcomm set includes industrial workloads. The SPEC CPU 2006 and SPEC CPU 2017 benchmark suites contain general-purpose applications. The GAP benchmark suite includes six graph processing kernels: (i) breadth-first search (bfs), (ii) page rank (pr), (iii) connected components (cc) [253], (iv) betweenness centrality (bc) [90], (v) triangle count (tc), and (vi) single-source shortest paths (sssp) [194]. For each kernel, we consider five different input graphs featuring different sizes and distributions of node degrees, presented in Table 3.4. Our evaluation reports results for

| Suite | Benchmarks |
|---|---|
| **SPEC CPU 2006** | gcc, mcf, milc, cactusADM, GemsFDTD, astar, sphinx3, xalancbmk, omnetpp |
| **SPEC CPU 2017** | mcf_s, xalancbmk_s, omnetpp_s |
| **Qualcomm (QMM)** | 125 floating point, integer, and server traces |
| **GAP** | bfs.kron, bfs.road, cc.urand, cc.twitter, bc.twitter, bc.kron, sssp.twitter, sssp.web, tc.twitter, pr.twitter, pr.kron |
| **XSBench** | xs.hash, xs.nuclide |

Table 3.5: Complete set of workloads used for evaluation.

the two input graphs that produce the highest pressure on the TLB hierarchy per kernel (if any). XSBench is evaluated using all different grid types (unionized, hash, nuclide), and we present results for the two most TLB intensive ones. We refer to GAP and XSBench workloads as Big Data (BD) workloads because they have massive memory footprints [298]. For the rest of this chapter, we use QMM, SPEC, and BD to refer to the Qualcomm, the SPEC CPU 2006 and SPEC CPU 2017, and the GAP plus XSBench workloads, respectively.

Workloads with a TLB MPKI rate of at least 1 are considered TLB intensive and thus taken into account in our evaluation. After the MPKI selection, our set of workloads includes 125 QMM workloads, 12 SPEC CPU workloads, and 13 BD workloads. All traces were obtained using the SimPoint [218] methodology. Table 3.5 presents the complete set of workloads used for evaluating our proposals.[1]

Each SPEC and BD workload runs 250 million warmup instructions and one billion instructions are executed to measure the experimental results. For the QMM workloads we use 50 million warmup instructions and 100 million instructions for measuring the results, similar to prior work [196].

[1]Section 3.8.2.1 presents evaluation considering the entire SPEC CPU 2006 and SPEC CPU 2017 benchmark suites to highlight that our proposals do not harm the performance of non TLB intensive workloads.

## 3.8 Experimental Campaign

This section presents our experimental campaign. Section 3.8.1 quantifies the benefits of the Sampling-Based Free TLB Prefetching (SBFP) on prior and novel TLB prefetchers and Section 3.8.2.1 focuses on Agile TLB Prefetcher (ATP) coupled with SBFP.

### 3.8.1 Impact of SBFP

To demonstrate that exploiting PTE locality can positively impact TLB prefetching while highlighting the benefits of the proposed SBFP scheme, we combine all prior TLB prefetchers (SP, DP, ASP) and new TLB prefetchers (STP, H2P, MASP, and ATP) with SBFP as well as the NoFP, NaiveFP, and StaticFP scenarios presented in Section 3.7.3. Subsequent sections examine the impact of these scenarios using multiple metrics and comment on the results, providing the key conclusions of our experimental campaign.

#### 3.8.1.1 TLB Coverage and Performance

Figures 3.12 and 3.13 quantify the impact of the different scenarios that apply free prefetching on TLB prefetching by presenting the TLB MPKI reduction and the geometric mean speedups experienced by state-of-the-art and new TLB prefetchers, respectively. This set of experiments assumes a 64-entry PB; we comment on the impact of different PB sizes in the end of this section. The baseline system does not apply prefetching at any TLB level.

**Coverage**

Looking at Figure 3.12, we observe that all considered TLB prefetchers benefit from free TLB prefetching since they achieve higher TLB MPKI reductions when they are combined with the NaiveFP, StaticFP, and SBFP schemes than when free TLB prefetching is not exploited (NoFP). We observe this behavior because (i) the free prefetches provide PB hits that eliminate demand page walks due to the reduction of TLB misses, and (ii) most of the prefetch requests issued by the TLB prefetchers have already been prefetched for free, avoiding prefetch page walks that waste useful resources (*e.g.*, page table walker ports, energy). Notably, SBFP provides significantly higher MPKI reduction rates than the other evaluated scenarios (NoFP, NaiveFP, StaticFP) when combined with the different TLB prefetchers. For

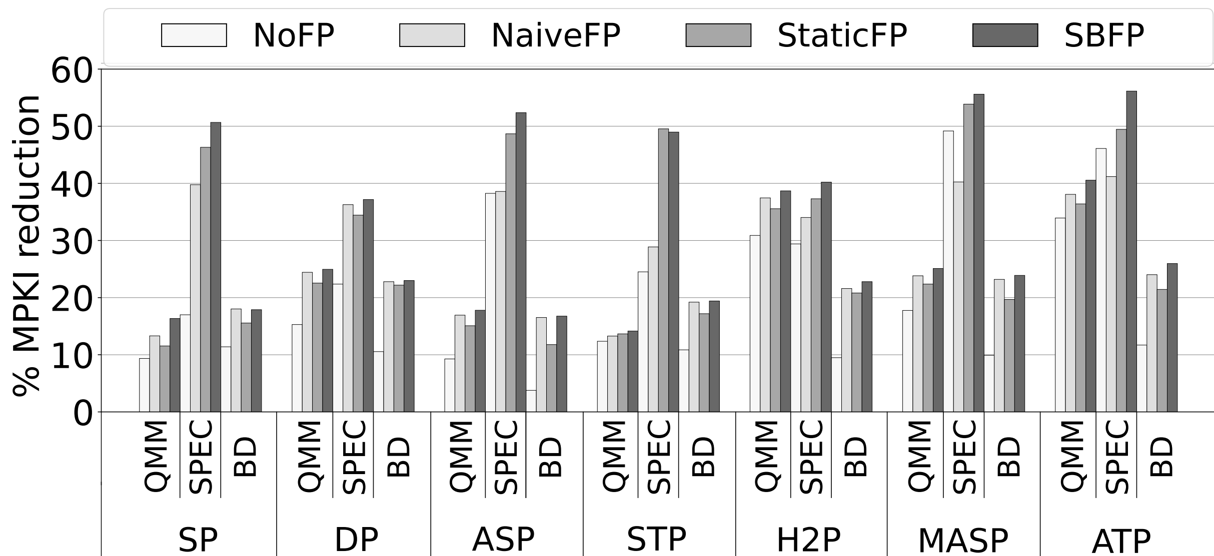


Figure 3.12: TLB MPKI reduction offered by different TLB prefetchers (SP, DP, ASP, STP, H2P, MASP, ATP) when combined with different scenarios that exploit free TLB prefetching (NoFP, NaiveFP, StaticFP, SBFP). Higher is better. The QMM, SPEC, and BD workloads experience 13.9, 3.4, and 38.9 average TLB MPKI in the baseline that does not apply TLB prefetching, respectively.

example, ATP coupled with SBFP reduces the TLB MPKI from 13.9 to 8.2 (41% reduction) for the QMM workloads, from 3.4 to 1.46 (56% reduction) for the SPEC workloads, and from 38.9 to 29.2 (25% reduction) for the BD workloads. Stress that the lowest TLB MPKI reduction is achieved by the NoFP scenario that does not exploit free TLB prefetching; this is the case for all considered TLB prefetchers. The reason for such behavior is that the NoFP scenario does not store any of the free PTEs into the PB, thus missing the opportunity to reduce the number of TLB misses.

#### Performance

The performance evaluation, presented in Figure 3.13, justifies the TLB MPKI reduction rates and provides conclusions consistent with the ones presented above: (i) all TLB prefetchers perform better when free prefetching is enabled compared to no free prefetching (NoFP), and (ii) SBFP is the optimal choice between the scenarios that exploit free prefetching (NaiveFP, StaticFP). The main conclusion drained out of this evaluation is that all TLB prefetchers experience their highest (lowest) speedups when combined with SBFP (NoFP). For instance, SP with SBFP improves performance by 5.6% over SP with NoFP for the SPEC

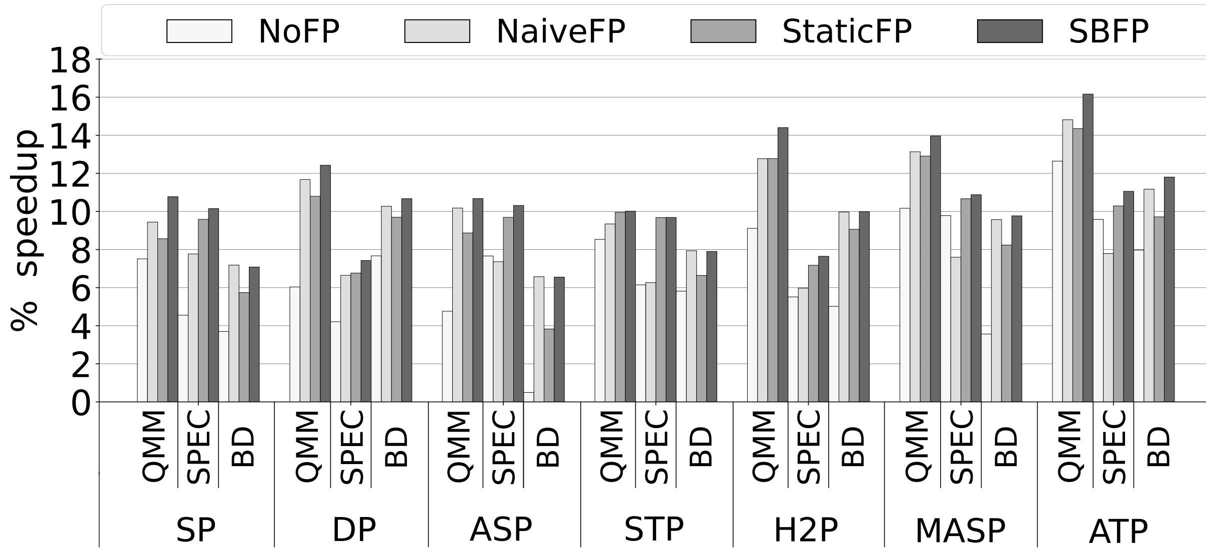


Figure 3.13: Geometric mean speedups delivered when different TLB prefetchers (SP, DP, ASP, STP, H2P, MASP, ATP) are combined with different scenarios that exploit free TLB prefetching (NoFP, NaiveFP, StaticFP, SBFP). Higher is better.

workloads. Notably, MASP outperforms ASP with and without exploiting free prefetching because MASP is capable of capturing more generic patterns than ASP. This verifies the value of the modifications we propose in the design of ASP. Overall, ATP with SBFP yields a geometric mean speedup of 16.2%, 11.1%, and 11.8% for the QMM, SPEC, and BD workloads, respectively.

### Best Performing Configuration

Next, we compare ATP coupled with SBFP, with the best prior TLB prefetcher (it is not always the same across benchmark suites and scenarios that exploit free prefetching). ATP with SBFP outperforms the best state-of-the-art TLB prefetcher with NoFP by 8.7%, 3.4%, and 4.2% for the QMM, SPEC, and BD workloads, respectively. Moreover, ATP with SBFP improves performance over the best state-of-the-art TLB prefetcher with NaiveFP by 4.6%, 3.4%, and 1.6% for the QMM, SPEC, and BD workloads, respectively. Finally, ATP with SBFP outperforms the best prior TLB prefetcher with StaticFP by 5.4%, 1.4%, and 2.1% for the QMM, SPEC, and BD workloads, respectively. The main takeaway of this comparison is that ATP coupled with SBFP provides the overall best performance across all possible combinations of prior TLB prefetchers and techniques that exploit free TLB prefetching.

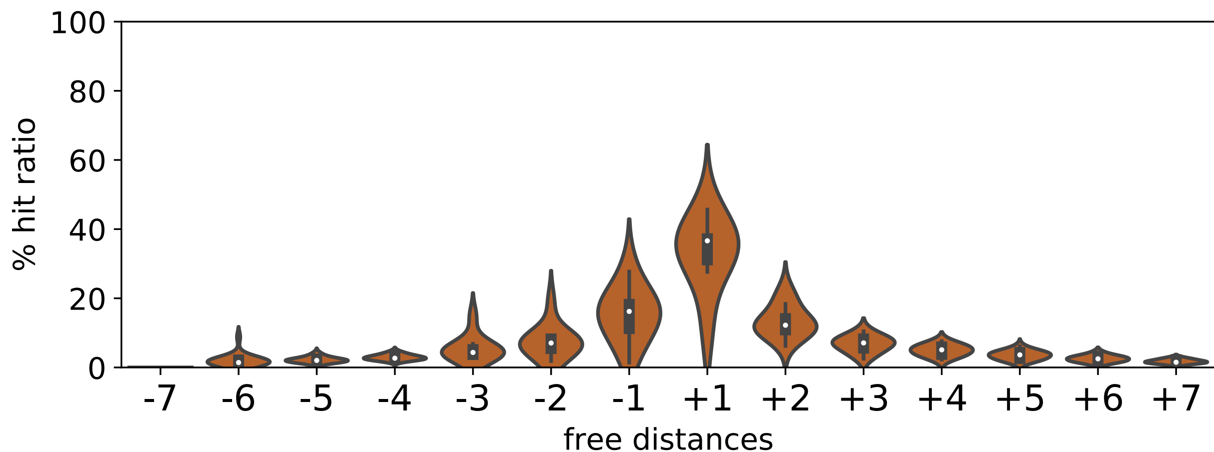


Figure 3.14: Normalized hit ratio of each free distance depicted with violin plots, across all considered workloads and TLB prefetchers.

#### Insights on SBFP's Superior Performance

Figure 3.13 shows that the NaiveFP scenario outperforms the StaticFP scenario for the QMM and BD workloads. For the SPEC workloads, the opposite behavior is observed. This happens because StaticFP always uses the overall most useful free distances based on static offline exploration, but cannot use the non-selected free distances that are seldom beneficial in specific execution phases. In these cases, NaiveFP outperforms StaticFP because it fetches all available free PTEs into the PB. The main disadvantage of NaiveFP over StaticFP is that it does not examine the usefulness of the free prefetches, resulting in suboptimal performance enhancements due to PB thrashing. This analysis was the core inspiration for the design of the SBFP scheme that identifies the most useful free PTEs per execution phase, combining the advantages of NaiveFP and StaticFP.

To further demonstrate the benefits of SBFP over the other scenarios that exploit PTE locality, Figure 3.14 illustrates the usefulness of the different free distances by presenting the aggregated distribution of the normalized hit ratio per free distance among all prefetchers and considered workloads. Free distances -2, -1, +1 and +2 produce the majority of the PB hits due to free prefetching, verifying the quality of the statically selected free distances per TLB prefetcher, presented in Table 3.3, because the optimal set of static free distances of all TLB prefetchers is generally among the overall best free distances. However, some free distances (*e.g.* -3, +3, -6) significantly contribute to the reduction of TLB misses for a large subset of benchmarks. The key conclusion arising from this analysis is that different free

distances are useful for different TLB prefetchers and for different execution phases, which reveals the reason why SBFP exploits best the chance for free TLB prefetching compared to the other scenarios that exploit PTE locality.

**Impact of PB Size**

To justify why we consider a 64-entry PB, we evaluate scenarios with various PB sizes. Our experiments reveal that using PBs with 16 and 32 entries provides lower performance compared to a 64-entry PB. Specifically, ATP coupled with SBFP experiences a 56% and 32% performance reduction using a PB with 16 and 32 entries for the SPEC workloads, respectively. We observe similar behavior for the QMM and BD workloads. In addition, larger PBs provide negligible performance improvements with respect to a 64-entry PB. Besides, the PB lookup takes place on the critical path of accessing memory, thus, TLB prefetching is fundamentally limited by the PB size. Therefore, a 64-entry PB is a good design point. Subsequent sections consider a 64-entry PB, unless stated otherwise.

#### 3.8.1.2 Cost of TLB prefetching

This section quantifies the impact of PTE locality exploitation on the cost of hardware TLB prefetching. To do so, we present in Figure 3.15 the normalized number of memory references triggered by page walks (demand plus prefetch) for all considered scenarios that exploit free prefetching (NoFP, NaiveFP, StaticFP, SBFP) and TLB prefetchers (SP, DP, ASP, STP, H2P, MASP, ATP), similar to Figure 3.3 in Section 3.3. The term *memory reference* refers to a page walk reference that is served by the memory hierarchy (Section 2.4.1.1); note that our evaluation takes into account cache locality in page walk memory references (Section 3.7). The normalization factor, 100% in Figure 3.15, is the total number of memory references for demand page walks without TLB prefetching.

Looking at Figure 3.15, we observe a large increase in page walk memory references when free prefetching is not exploited (NoFP). This happens because all prefetches require a prefetch page walk to fetch the requested PTEs into the PB and the reduction of the demand page walks is smaller than the number of prefetch page walks introduced. The only exception is the ASP prefetcher that introduces only 1% more page walk memory references over the baseline that does not employ TLB prefetching; the state field of ASP's prediction is responsible for such low overhead, as explained in Section 3.3. However, ASP

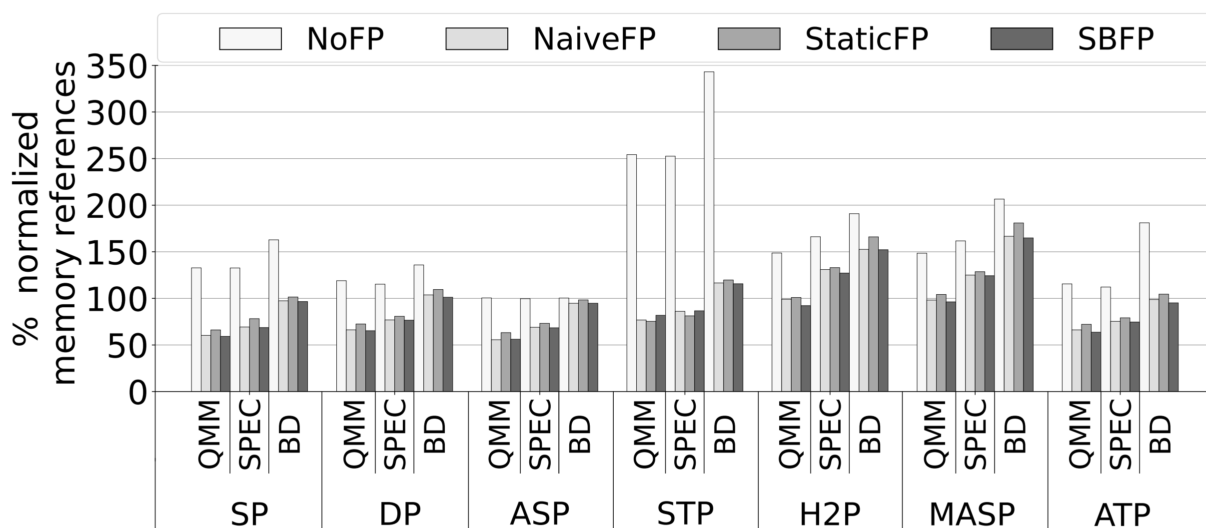


Figure 3.15: Normalized memory references due to page walks (demand and prefetch) across all considered TLB prefetchers and scenarios that exploit page table locality. Lower is better.

combined with NoFP provides poor coverage and speedup results, as shown in Figures 3.12 and 3.13. Focusing on the BD workloads, SP, DP, ASP, STP, H2P, MASP, and ATP trigger 63%, 36%, 1%, 250%, 90%, 106%, and 81% additional memory references compared to the scenario without TLB prefetching, respectively. We observe similar behavior for the SPEC and QMM workloads.

All techniques that exploit free TLB prefetching significantly reduce the number of page walk memory references because (i) the majority of the prefetches that would otherwise require a prefetch page walk are proactively fetched in the PB as free prefetches, and (ii) free prefetches provide PB hits that save long-latency demand page walks. All considered TLB prefetchers experience their highest reduction in terms of memory references when combined with the SBFP scheme. This behavior occurs because SBFP saves more TLB misses than NaiveFP and StaticFP, *i.e.*, it eliminates more demand page walks, as presented in Figures 3.12 and 3.13. Overall, ATP with SBFP eliminates by 37%, 26%, and 5% the number of memory references due to page walks compared to the scenario without TLB prefetching for the QMM, SPEC, and BD workloads respectively. Finally, we observe a huge reduction in memory references when free prefetching is exploited compared to the scenario without free prefetching (NoFP). For instance, ATP with SBFP requires 38%, 52%, and 86% fewer memory references for page walks compared to NoFP for the QMM, SPEC, and BD workloads respectively.

For readability, Figure 3.15 shows the normalized number of memory references triggered by both demand and prefetch page walks. However, memory references that hit in the cache hierarchy incur lower latency than the ones going to DRAM, and the latency cost of a demand page walk is more critical than the latency of a prefetch page walk since the former takes place in the critical path of accessing memory while the latter operates in the background. Section 3.8.2.2 elaborates on these implications.

## 3.8.2 Evaluation of ATP coupled with SBFP

This section compares ATP coupled with SBFP against the state-of-the-art TLB prefetchers, the constituent prefetchers of ATP, and other approaches that improve TLB performance while showing the benefits of our proposal when large pages are used. All prefetchers use a 64-entry PB for the reasons outlined in Section 3.8.1.1.

### 3.8.2.1 Performance Comparison

**Comparison with Prior TLB Prefetchers**

Figure 3.16 presents the performance of SP, DP, ASP, and ATP combined with SBFP for all evaluated workloads; The evaluation of the QMM, BD, and SPEC workloads is presented in Figure 3.16 (top), Figure 3.16 (middle), and Figure 3.16 (bottom), respectively. For the SPEC workloads, we also show the geometric mean for the whole suite (GM_All in Figure 3.16), including also the non TLB intensive workloads. To demonstrate that our proposals do not harm the performance of non TLB intensive workloads, Figure 3.17 presents the performance impact on all SPEC workloads, no matter their TLB MPKI. Despite that most of the non TLB intensive workloads from SPEC CPU suites exhibit around zero TLB MPKI rates, there are some workloads (*e.g.* `soplex`, `libquantum` in Figure 3.17), that experience around 0.9 TLB MPKI. These workloads are not included in our evaluation because they do not reach the TLB MPKI threshold of 1, as explained in Section 3.7.4. However, applying TLB prefetching for these workloads provides small performance benefits (<4%) due to their modest TLB MPKI rates; this is the reason why GM_All is lower than GM_Inte in Figures 3.16 (bottom) and Figure 3.17.

Focusing on the TLB intensive workloads, ATP combined with SBFP outperforms all state-of-the-art TLB prefetchers, achieving geometric mean speedups over the best already

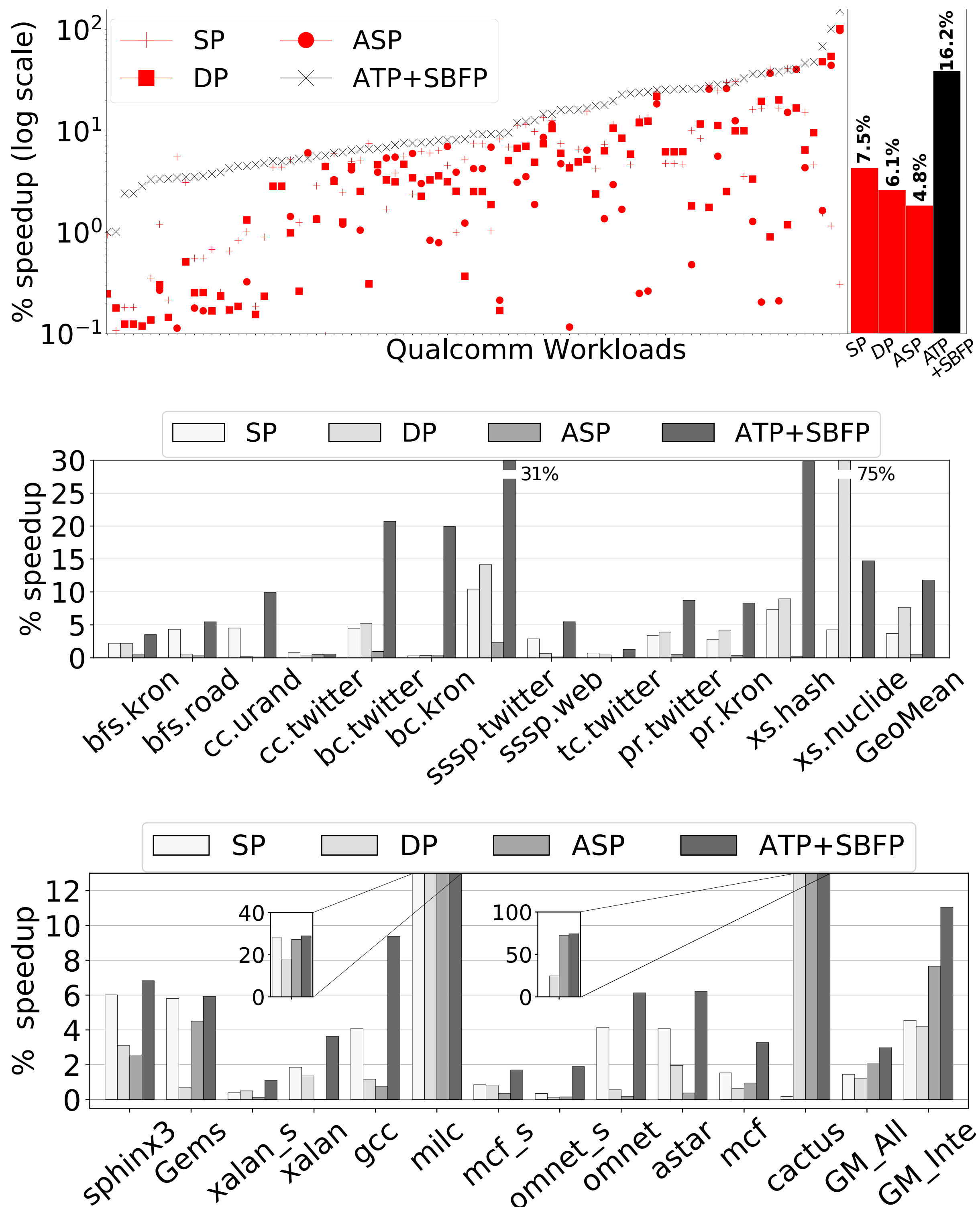


Figure 3.16: Performance comparison between ATP coupled with SBFP and previously proposed TLB prefetchers (SP, DP, ASP). The baseline does not employ TLB prefetching. The evaluation of the QMM, BD, and SPEC workloads is presented in the top, middle, and bottom subfigures, respectively. Higher is better.

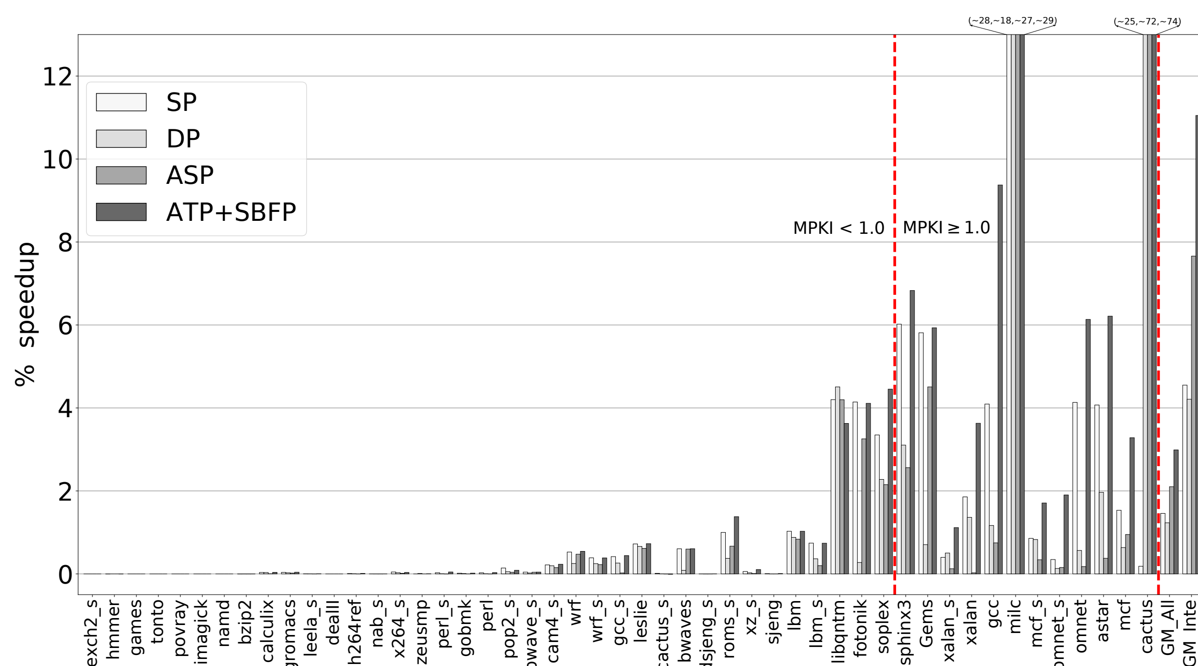

Figure 3.17: Performance comparison between ATP coupled with SBFP and previously proposed TLB prefetchers (SP, DP, ASP), considering the entire SPEC CPU 2006 and SPEC CPU 2017 benchmark suites. Higher is better.

proposed TLB prefetcher per benchmark suite of 8.7%, 3.4%, and 4.2% for the QMM (top), SPEC (bottom), and BD (middle) workloads, respectively. Moreover, we observe that for workloads such as `xs.nuclide` and `sssp.twitter` DP provides high performance gains since they exhibit great distance correlation. For the `xs.nuclide`, DP outperforms our proposal since ATP enables H2P when distance correlation is observed, although DP is capable of detecting more complex distance patterns compared to H2P.

### Deep Dive Into ATP+SBFP

Figure 3.13 demonstrates that ATP significantly outperforms its constituent prefetchers (STP, H2P, MASP). This happens because ATP efficiently combines these prefetchers by selecting the most appropriate prefetcher per TLB miss while disabling TLB prefetching during execution phases that it is not useful. To validate our statements, Figure 3.18 shows the fraction of TLB misses that ATP enables each TLB prefetcher or disables prefetching. When none of the prefetchers is capable of capturing the TLB access patterns of the SPEC work-

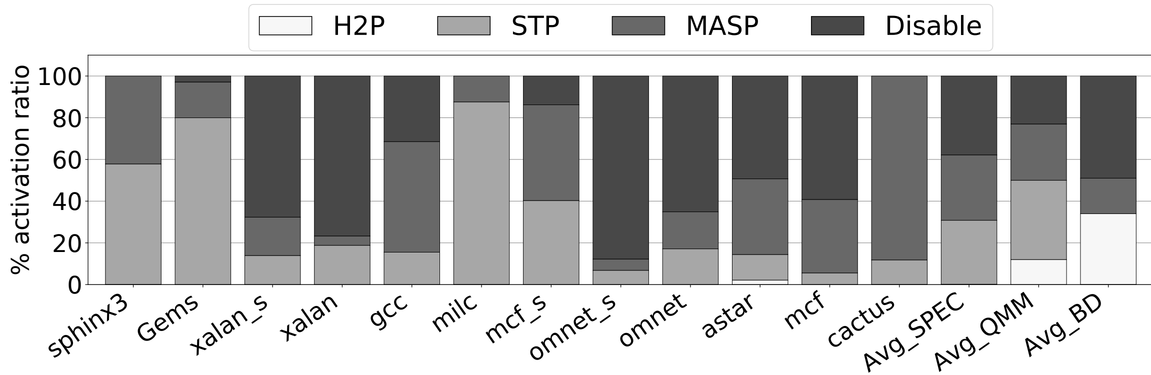


Figure 3.18: Fraction of TLB misses that ATP enables H2P, STP, MASP or disables TLB prefetching.

loads, ATP disables prefetching (*e.g.,* `xalan_s`, `mcf`). For benchmarks with strided patterns (*e.g.,* `milc`), ATP enables mostly STP. MASP is enabled only when the TLB miss stream correlates well with the PC (*e.g.,* `cactus`, `mcf_s`). As explained in Section 3.5, ATP enables H2P only when it is confident that H2P will produce useful prefetches because H2P uses large distances that may pollute the PB content in case of inaccurate prefetches. Figure 3.18 reveals that the SPEC workloads are not benefited by the observed distance correlation, thus ATP never enables H2P. Contrarily, a large number of the QMM and BD workloads (*e.g.,* `sssp.twitter`, `xs.nuclide`), experience TLB misses that exhibit great distance correlation. Consequently, ATP enables H2P 12% and 34% of the time for the QMM and BD workloads, respectively. Specifically, for `xs.nuclide` ATP selects H2P for prefetching 61% of the time, explaining why DP outperforms ATP for this workload. This evaluation verifies that combining ATP with SBFP successfully addresses all the findings of Section 3.3.

The next question we quantitatively answer is how much is the contribution of each module (ATP, SBFP) on the performance enhancements of our proposal? To provide a clear answer, Figure 3.19 presents a breakdown of the normalized number of PB hits provided by our proposal across the considered benchmark suites, *i.e.*, it shows the fraction of PB hits provided by the ATP and the SBFP modules. Moreover, Figure 3.19 breaks down the PB hits provided by ATP into three subcategories; PB hits provided by the constituent prefetchers of ATP. On average, prefetch requests issued by ATP are responsible for 60%, 56%, and 41% of the PB hits regarding the QMM, SPEC, and BD, respectively. SBFP's contribution is also significant since it is responsible for 40%, 44%, and 59% of the total PB hits for the QMM, SPEC, and BD workloads, respectively. The main takeaway is that both ATP and SBFP play an equally significant role in achieving significant performance enhancements.

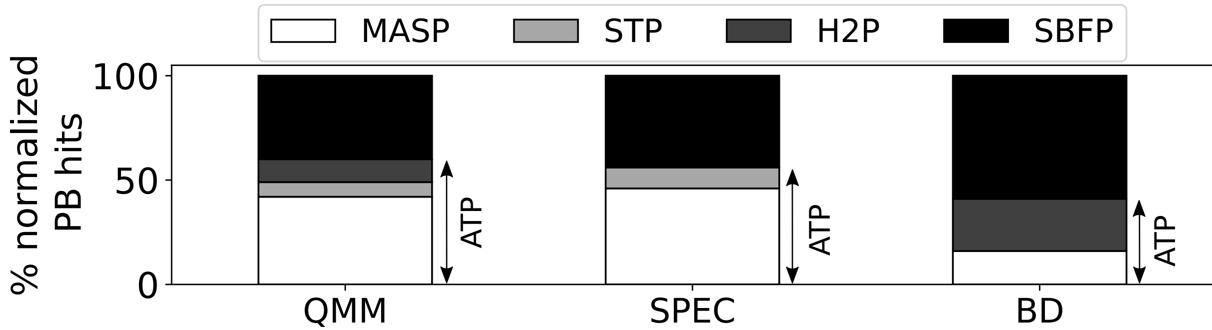


Figure 3.19: Percentage of PB hits provided by ATP (its constituent prefetchers) and SBFP.

#### 3.8.2.2 Cost of TLB Prefetching

This section elaborates on the page walk footprint of our proposal, ATP combined with SBFP. Figure 3.20 (up) presents the distribution of the normalized number of memory references due to page walks (demand plus prefetch) out of the total number of memory references for demand page walks without TLB prefetching, across all benchmark suites. For the QMM workloads ATP with SBFP reduces memory references by 37%, while SP, DP, and ASP introduce 33%, 19%, and 1% additional page walk memory references over the baseline that does not apply TLB prefetching. We report similar results for the SPEC workloads. For the BD workloads, we observe a lower reduction in page walk memory references than SPEC and QMM workloads because the evaluated TLB prefetchers are unable to accurately detect highly irregular TLB miss patterns.

The distribution presented in Figure 3.20 (up) provides a coarse representation of our proposal's page walk memory footprint. Figure 3.20 (down) provides a finer analysis by presenting a breakdown of the average normalized number of page walk memory references (presented as a bullet inside the violins of Figure 3.20 (up)). Specifically, Figure 3.20 (down) shows (i) a breakdown of the memory references caused by demand and prefetch page walks, and (ii) a breakdown of the level of the memory hierarchy that serves the memory references of both demand and prefetch page walks. First, we observe that ATP with SBFP provides the highest reduction in demand page walks for all considered suites since it enables the right prefetcher per TLB miss, thus providing more PB hits than the prior TLB prefetchers. In addition, ATP with SBFP triggers fewer memory references due to prefetch page walks than the previously proposed TLB prefetchers because (i) it exploits SBFP to prefetch PTEs that otherwise would need a separate prefetch page walk to be fetched into

the PB, (ii) the throttling mechanism of ATP disables TLB prefetching during execution phases that is not helpful, and (iii) for strided patterns the selection mechanism of ATP enables STP, which uses small strides that are mostly served by free prefetches. Furthermore, ATP combined with SBFP provides the highest reduction in memory references that require accessing DRAM among the evaluated TLB prefetches. ATP with SBFP drastically reduces the DRAM accesses of demand page walks, which provides great performance benefits, at the cost of introducing a few DRAM accesses for prefetch page walks, which are not placed in the critical path. The bottom line is that our proposal reduces the number of memory references due to page walks, their cost, and the performance penalties they cause.

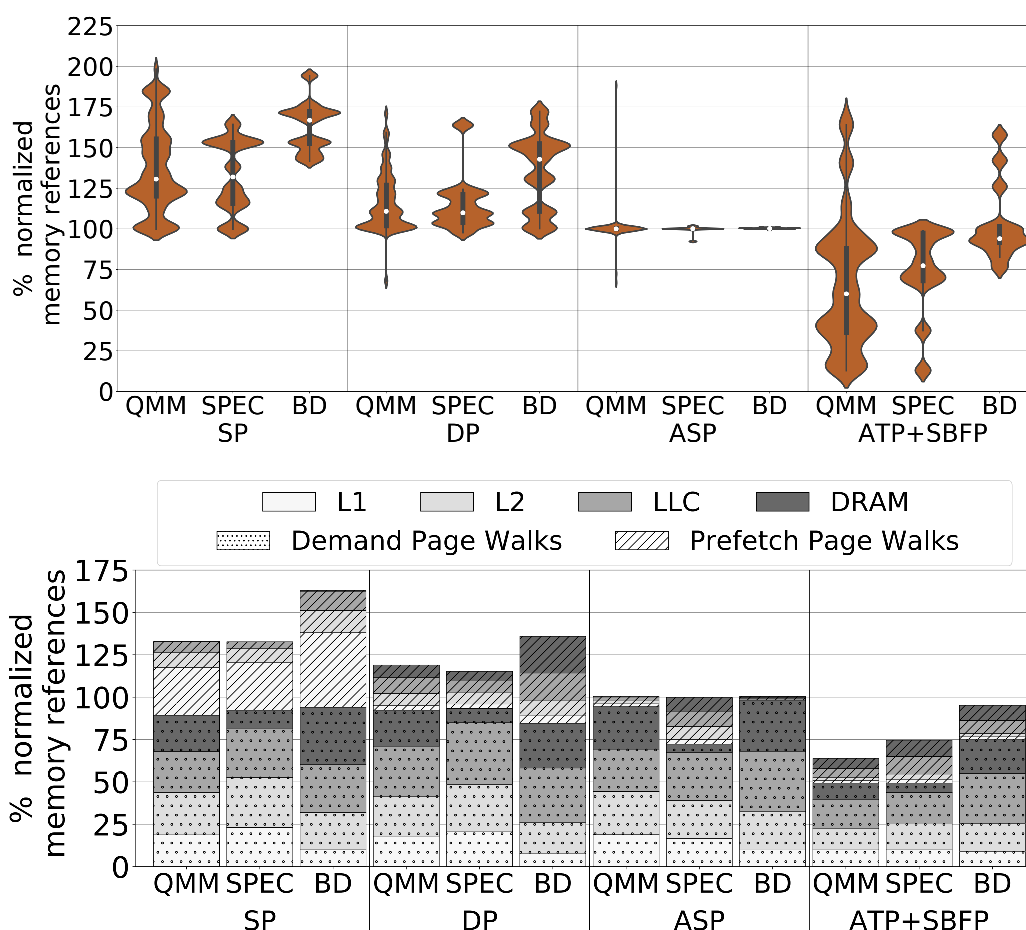


Figure 3.20: Distribution of the normalized number of memory references due to page walks (demand plus prefetch) depicted with violin plots (up). Breakdown of the average memory references caused by demand and prefetch page walks depending on which level of the memory hierarchy serves these references (down). The baseline memory references correspond to 100%.

| TLB Prefetcher | Storage Budget |
|---|---|
| **SP** | 4928 bits $\simeq$ 0.60KB |
| **DP** | 7808 bits $\simeq$ 0.95KB |
| **ASP** | 12160 bits $\simeq$ 1.48KB |
| **STP** | 4928 bits $\simeq$ 0.60KB |
| **H2P** | 4992 bits $\simeq$ 0.61KB |
| **MASP** | 16960 bits $\simeq$ 1.47KB |
| **ATP** | 18752 bits $\simeq$ 1.68KB |
| **SBFP** | 2700 bits $\simeq$ 0.33KB |

Table 3.6: Storage budget of the considered TLB prefetchers presented in Section 3.7.2. The storage budgets of the TLB prefetchers include the storage of a 64-entry PB (4928 bits $\simeq$ 0.60KB). The budget of SBFP does not include a 64-entry PB since it is counted in the budget of ATP; our proposal (ATP+SBFP) uses a shared PB.

#### 3.8.2.3 Storage Budget

Table 3.6 presents the storage budget of all evaluated TLB prefetchers, including the storage budget of a 64-entry PB. Each PB entry requires 36 bits for storing the virtual page number, 36 bits for the physical page number, and 5 attribute bits. Therefore, a 64-entry PB requires 0.60KB of storage to be implemented. Regarding the prefetchers, SP and STP are stateless. H2P requires only 64 bits for storing the previously observed distance. ASP stores 60 bits for the PC, 36 bits for the virtual page number, 15 bits for stride, and 2 bits for the state field. DP stores distances of 15 bits. MASP stores 60 bits for the PC, 36 bits for the virtual page number, and 15 bits for the corresponding stride. Each FPB entry of ATP stores 36 bits for the virtual page number. Therefore, considering a 64-entry PB, SP, DP, ASP, and ATP require in total 0.60KB, 0.95KB, 1.48KB, and 1.68KB to be realized, respectively. ATP is slightly more expensive than the previously proposed TLB prefetchers. Regarding the storage cost of SBFP, each Sampler entry requires 36 bits for the virtual page number plus 4 bits for the corresponding free distance and the FDT uses a 10-bit counter per free distance. Therefore, SBFP requires 0.33KB to be implemented. In total, ATP combined with SBFP requires 2.01KB of storage to be implemented. The benefits of our proposal make this minimal overhead affordable for a realistic design.

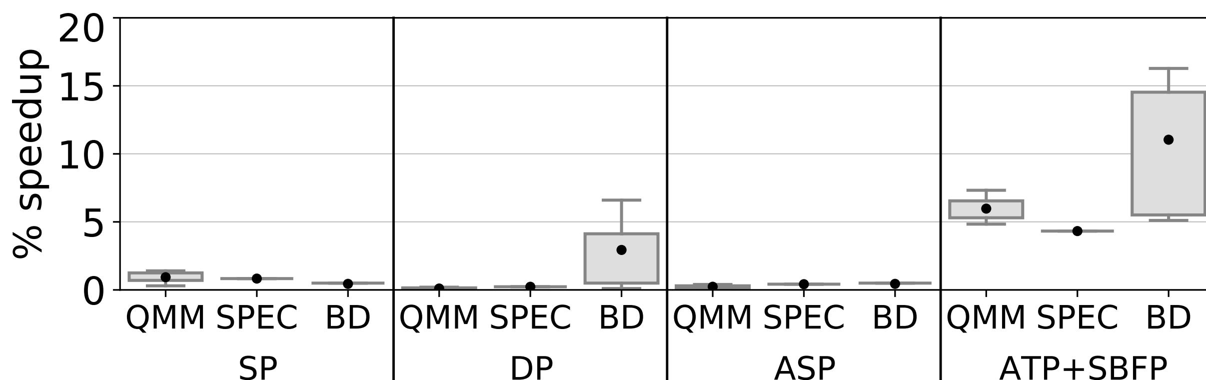


Figure 3.21: Distribution of speedups delivered by ATP coupled with SBFP and previously proposed TLB prefetchers (SP, DP, ASP) when the baseline system uses only 2MB pages and does not employ TLB prefetching. Higher is better.

#### 3.8.2.4 Large Pages

To examine the impact of large pages on our proposal, we evaluate ATP combined with SBFP and the state-of-the-art TLB prefetchers using a baseline with only 2MB pages, similar to prior work [131, 188]. We observe significant TLB MPKI reduction for most of the considered workloads when 2MB pages are used, although a few of them still experience high TLB MPKI rates. For these workloads, we measured that ATP with SBFP reduces by 88% on average the TLB misses that 2MB pages cannot eliminate. Figure 3.21 shows the performance of SP, DP, ASP, and ATP combined with SBFP for the 2MB scenario. The baseline implies using 2MB pages without TLB prefetching. ATP with SBFP provides a geometric mean speedup of 5.1%, 4.3%, and 9.9% for the QMM, SPEC, and BD workloads that still experience significant TLB MPKI rates, respectively. Regarding the previously proposed TLB prefetchers, SP, DP, and ASP provide negligible speedups, except DP which achieves notable performance gains for the BD workloads since the TLB miss patterns of these workloads exhibit great distance correlation. Note that the SPEC set of workloads contains only one benchmark, `mcf`, since for the other SPEC workloads the 2MB pages eliminate the majority of the observed TLB misses. Finally, the vast majority of the PB hits (89% on average) are produced by free prefetches because free TLB prefetching with 2MB pages covers a larger amount of memory compared to standard 4KB pages.

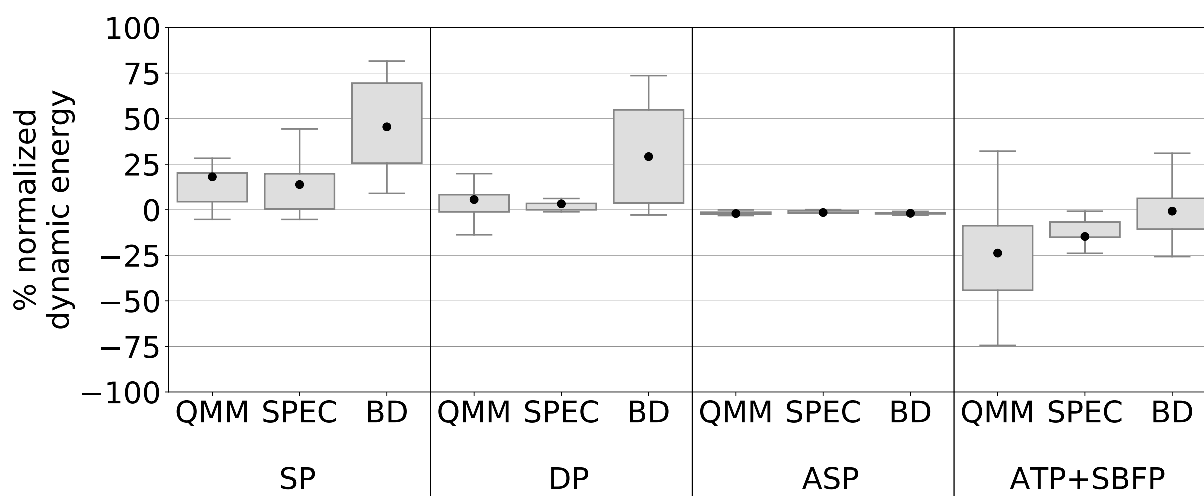


Figure 3.22: Impact of ATP with SBFP and previously proposed TLB prefetchers (SP, DP, ASP) on the dynamic energy consumption of address translation depicted with box plots. Lower is better.

#### 3.8.2.5 Energy Consumption

This section quantifies the impact of our proposal on the dynamic energy consumed by the virtual memory subsystem. To measure the baseline dynamic energy consumption of address translation we take into account all accesses into the iTLB, dTLB, L2 TLB, MMU-Caches as well as all page walk references to the memory hierarchy. When a TLB prefetcher is used the dynamic energy is reduced by saving demand page walks due to PB hits but it is also increased by the accesses in the PB, the Sampler, the FDT, and the triggered references to memory hierarchy for prefetch page walks.

Figure 3.22 presents the dynamic energy consumed by address translation when ATP coupled with SBFP and the state-of-the-art TLB prefetchers operate in the system. ATP with SBFP lowers dynamic energy by 24%, 14.6%, and 1% for the QMM, SPEC, and BD workloads, respectively. SP, DP, and ASP increase the dynamic energy usage, especially for the BD workloads. We remark on this behavior because our proposal saves demand page walks by hitting in the PB and also decreases the number of prefetch page walks by leveraging the SBFP module. Regarding the static energy consumption of address translation, negligible impact is observed.

### 3.8.3 Comparison with Other Approaches

This section compares ATP coupled with SBFP against other state-of-the-art techniques that improve TLB performance. Figure 3.23 presents the experimental results.

#### ISO Storage

We compare our proposal against a system that does not employ prefetching for the TLB hierarchy that, for fairness, has an enlarged L2 TLB. Specifically, the L2 TLB of this system is augmented with 265 entries without affecting its access time, matching the storage budget of ATP and SBFP (1.68KB + 0.33KB). Figure 3.23 shows that ATP combined with SBFP outperforms this scenario by 14.7%, 9.8%, and 11.5% for the QMM, SPEC, and BD workloads, respectively.

#### Free Prefetching into the TLB

Prior work [86] leverages PTE locality to fetch all free PTEs directly into the TLB upon demand page walks. This approach does not use TLB prefetchers nor PBs, and it does not consider selecting only the useful free PTEs per page walk. Figure 3.23 shows that this approach (FP-TLB) reduces performance by 10.2% and 7.8% for the QMM and SPEC workloads, respectively, due to the eviction of useful PTEs from the TLB. Our results are consistent with prior work [85, 165] stating that placing all the free PTEs directly to TLB may pollute its content. There is no previous work that stores prefetches directly into the TLB using big data workloads. Our evaluation reveals that placing all free PTEs into the TLB increases performance by 5.2% for the BD workloads. These workloads experience massive TLB MPKI rates and thrash the TLB. Therefore, storing useful free PTEs in the TLB improves performance for big data applications. Still, ATP coupled with SBFP outperforms this scenario. Finally, we observe similar behavior when our proposal places all the prefetches, not only the free prefetches, directly into the TLB.

#### Recency-based TLB Preloading [243]

This is a software approach that modifies the page table so that each PTE stores the virtual page that is subsequently accessed. A fundamentally similar approach that only requires $\mu$architectural modifications is Markov prefetching [158]. A hardware Markov prefetcher

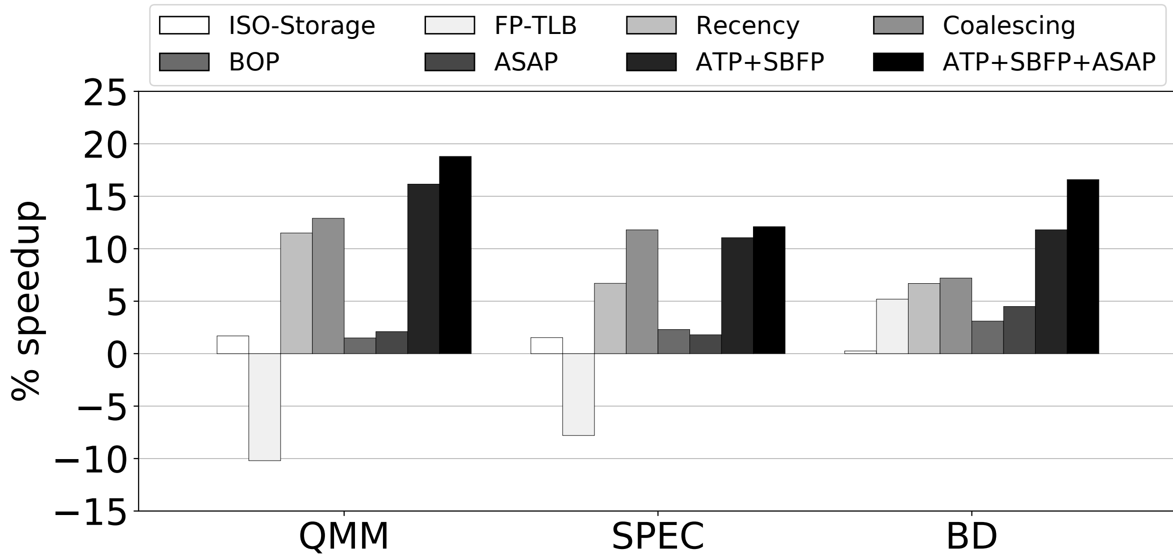


Figure 3.23: Performance comparison of ATP combined with SBFP with other approaches that improve TLB performance. Higher is better.

consists of a prediction table indexed with the virtual page where each entry contains a virtual page that is predicted to be accessed next, as described in Section 2.4.1.2. To approximate the behavior of Recency-based TLB Preloading, we enhance a Markov prefetcher with a 64K-entry prediction table. Figure 3.23 reveals that our proposal outperforms this approach by 4.7%, 4.4%, and 4.3% for the QMM, SPEC, and BD workloads, respectively. This approach requires a very large hardware budget, infeasible for a realistic design.

#### TLB Coalescing

Coalescing approaches [219, 221, 270] rely on the contiguity of both virtual and physical memory and provide limited benefits when contiguity is absent (*e.g.,* due to fragmentation). Contrarily, SBFP exploits virtual address space contiguity and ATP relies only on the memory access patterns of the application. Therefore, our proposal is orthogonal to TLB coalescing. However, we compare ATP coupled with SBFP against a scenario with perfect contiguity in the virtual and physical memory where each TLB entry stores 8 adjacent PTEs. Figure 3.23 shows that this scenario improves performance by 12.9%, 11.8%, and 7.2% for the QMM, SPEC, and BD workloads, respectively. This scenario delivers great performance since it increases the TLB reach. Still, our proposal outperforms this scenario for the QMM and BD workloads, while the difference for the SPEC workloads is negligible.

### Data Cache Prefetching

Cache prefetchers typically try to learn strides within 4KB physical pages [172, 195]. Consequently, the number of observed strides is limited. TLB prefetchers try to capture varying stride patterns since a prefetched page might be far from the page that triggered the TLB miss. Therefore, using data cache prefetchers for data TLB prefetching intuitively limits the TLB miss patterns that can be captured. To justify this observation, we convert Best-Offset-Prefetcher (BOP) [195], a state-of-the-art data cache prefetcher, to prefetch for the TLB miss stream. We select BOP because it bears some similarity to ATP coupled with SBFP as both approaches try to identify the most useful strides per execution phase. We enrich the set of deltas that BOP uses with negative ones (the original version of BOP uses only positive deltas) to make sure that we do not underestimate its potential for TLB prefetching.

Figure 3.23 shows that when BOP is used as a TLB prefetcher, it improves performance by 2.3%, 1.5%, and 3.1% for the QMM, SPEC, and BD workloads, respectively. ATP with SBFP significantly outperforms BOP for all considered benchmark suites because BOP examines the effectiveness of a pre-defined set of deltas, thus it is unable to capture the varying stride TLB miss patterns, while our proposal captures more generic patterns; ATP activates the right TLB prefetcher per TLB miss and SBFP selects the most useful free PTEs per page walk. Moreover, BOP checks one offset per learning round, thus it requires several rounds to gain confidence for prefetching. In contrast, our proposal identifies faster the useful offsets as ATP leverages the operation of the FPBs to enable the most appropriate TLB prefetcher and SBFP learns the usefulness of all free PTEs concurrently. Finally, our proposal is more aggressive than BOP; ATP enables one prefetcher per TLB miss and SBFP uses all offsets that exceed a confidence threshold while BOP uses only the offset with the highest score.

### Prefetched Address Translation [188]

This work [188] proposes ASAP, a hardware scheme that lowers the page walk latency using direct indexing to prefetch deeper levels of the radix tree page table. Figure 3.23 shows that ASAP improves performance by 2.1%, 1.8%, and 4.5% for the QMM, SPEC, and BD workloads, respectively. ASAP provides important benefits when the MMU-Caches display low hit rates, but for workloads like SPEC and QMM which experience high MMU-Cache hit rates, its potential is limited. For the BD workloads, which have lower MMU-Cache hit

rates, ASAP provides significant performance benefits.

**Combining ATP, SBFP, and ASAP**

Hardware TLB prefetching is orthogonal to techniques aimed at lowering the page walk latency. Since ASAP lowers the latency cost of page walks, it can be also used to accelerate the prefetch page walks of ATP. Figure 3.23 shows that combining ATP with SBFP and ASAP improves geometric mean performance by 18.8%, 12.1%, and 16.6% for the QMM, SPEC, and BD workloads, respectively. ASAP increases the speedups of our proposal since it reduces the latency cost of both demand and prefetch page walks, thus the prefetched PTEs are fetched faster in the PB, improving the timeliness of TLB prefetching. To conclude, combining ATP, SBFP, and ASAP is a promising solution to the address translation performance bottleneck.

### 3.8.4 Interaction with OS Page Replacement Policy

As stated in Section 3.6, inaccurate TLB prefetches might harm the page replacement policy. A prefetch is harmful to the page replacement policy when it sets the access bit of the corresponding PTE, it is evicted from the PB without providing any hit, and it does not belong to the active footprint of the application. We measure that only 1.7%, 0.9%, and 3.6% of the prefetch requests of ATP with SBFP are harmful to the page replacement policy for the QMM, SPEC, and BD workloads. Therefore, we consider negligible the probability of negatively affecting the page replacement policy. We observe a small number of harmful prefetches because SBFP prefetches only the most useful free PTEs, while ATP dynamically enables the right TLB prefetcher and disables prefetching when it is not confident for issuing new prefetch requests. To avoid harming at all the page replacement policy, when a prefetch is proved to be useless our proposal could trigger a page walk in the background to reset the access bit of the corresponding translation in the page table. With this solution, the number of correcting page walks would be negligible due to the small number of harmful prefetches introduced by our proposal.[1] The main conclusion is that our proposal has a negligible impact on the page replacement policy.

[1] The correcting page walks could be issued only when the TLB MSHR is not full to avoid delaying any demand or prefetch page walk.

## 3.9 Related Work

### On the Locality of the Page Table

Prior work has identified locality of the PTEs in the last level of the radix tree page table. Pham *et al.* [219, 221] exploit this locality by modifying the TLB to increase its reach. These works require both virtual and physical contiguity while the proposed SBFP scheme solely exploits virtual contiguity. Liu *et al.* [183] use page table locality to increase the efficiency of the TLB. Bhattacharjee *et al.* [86] propose a shared last-level TLB organization for multi-core systems that fetch directly into the shared TLB the PTEs 1, 2, and 3 virtual pages away from the currently missing virtual page. Their technique only operates at demand page walks, without issuing prefetches that require prefetch page walks, and it targets multi-core systems with a shared last-level TLB. Shin *et al.* [255] exploit this locality by fetching all available PTEs for GPU applications. Wang [153] exploits page table locality to fetch free PTEs into a TLB buffer only for demand page walks, without providing any concrete implementation. Instead, we propose a practical implementation that dynamically exploits free TLB prefetching through sampling for multiple TLB prefetchers, we take advantage of page table locality for both demand and prefetch page walks, and our proposal improves the performance of private per-core TLBs using CPU applications.

### Other TLB Prefetching Schemes

Bhattacharjee *et al.* [85] propose two TLB prefetchers for multi-core systems, explained in Section 2.4.1.2. Both TLB prefetchers target parallel applications. The first is a TLB prefetching scheme that leverages TLB misses on common virtual pages among cores to prefetch the translations of those pages into the TLBs structures of the rest of the cores. The second TLB prefetcher exploits the DP prefetcher to save distance-predictable TLB misses across cores. ATP could form the base for the second aforementioned scheme.

### Reducing the TLB Miss Latency

One way to reduce the cost of TLB misses is by improving the performance of the MMU-Caches [66, 82]. Another approach is the POM-TLB [241], a large L3 TLB stored in the main memory that reduces the multiple memory references required by page walks to just one reference. DVMT [51] reduces the cost of TLB misses by allowing the application to

define the appropriate page table format so that fewer memory references are needed per page walk. Finally, hashed page tables [118, 140, 259, 300] have been proposed to resolve TLB misses faster than the radix page tables. Hardware TLB prefetching is an orthogonal approach as it eliminates TLB misses by converting them into PB hits.

#### Speculation Techniques

In speculation-based approaches [52, 65, 132, 222, 296], a missing translation is predicted, the processor continues executing instructions speculatively, and a validation page walk is performed in the background to validate the predicted address translation so that the page walk overlaps with useful work in case of correct speculation. Those approaches are affected by the system state (OS, load, fragmentation) as they depend on allocating contiguous virtual pages to contiguous physical pages to predict the missing address translations.

#### Increasing TLB Reach

Processor and OS vendors provide support for large pages [43] to increase TLB reach, as explained in Section 2.3.3. Prior work focuses on combining base and large pages in a performant and power-efficient way by using a single TLB [103, 212, 246] or separate TLBs [167] per page size. While large pages increase TLB reach and reduce the number of page walks, they are susceptible to performance issues when the OS cannot allocate such mappings (*e.g.,* due to memory fragmentation) or when the hardware support for large pages is limited with respect to the application needs. Several approaches try to bypass the limitations of large pages [71, 110, 119, 130, 169, 204, 213, 219, 267, 270, 271]. These approaches rely on leveraging the contiguity of mappings from the virtual to physical address space. Sub-blocked TLBs [270], CoLT [221], and Cluster TLB [219] target low-degree contiguity (*e.g.*, 8 pages), Hybrid Coalescing [214] targets medium-degree contiguity (*e.g.*, $< 512$ pages), while Direct Segments [71] and RMM [169] support single and multiple mappings of unlimited contiguity, respectively. Haria *et al.* [132] uses identity mappings for which the virtual address is the same as the physical address, and introduces support for region-level devirtualized access validation. These schemes are orthogonal to hardware TLB prefetching since they rely on explicit OS and hardware support to increase the effective capacity of the TLB while hardware TLB prefetching relies only on the memory access patterns of the application without any OS involvement.

### Virtualization

In virtualized environments each guest TLB miss requires a 2D page walk which can cause up to 24 memory references to be served [48]. Prior work [79, 120, 222] reduces the address translation overheads in virtualized environments, although, these overheads are still severe and undermine the performance of modern systems. Since the latency penalty of a TLB miss in virtualized systems is higher than 1D page walk latencies, our proposal, ATP coupled with SBFP, could potentially improve performance under virtualization, especially when free prefetching is exploited. However, in this case, SBFP has to be exploited in a 2D fashion which might incur additional complexity and logic overheads.

## 3.10 Summary

This chapter provides evidence that exploiting the locality in the last level of the radix tree page table for TLB prefetching purposes has the potential to provide significant performance and energy enhancements. To ameliorate the address translation performance bottleneck, we propose the *Sampling-Based Free TLB Prefetching (SBFP)*, a dynamic scheme based on sampling that identifies and prefetches only the most useful free PTEs per page walk, and the *Agile TLB Prefetcher (ATP)*, a composite data TLB prefetcher comprised of three low-cost engines while introducing adaptive selection and throttling schemes to enable the most appropriate TLB prefetcher per TLB miss and disable TLB prefetching when required. Considering an extensive set of contemporary industrial, academic, and big data workloads, we demonstrate that combining ATP with SBFP provides significant performance enhancements while reducing the vast majority of the page walk references to the memory hierarchy and the dynamic energy consumed by the virtual memory subsystem.

## 3.11 Future Work

The work presented in this chapter takes a fresh look at prefetching for the TLB (there was a paucity of research in the domain for more than 10 years) by evaluating and characterizing the previously proposed TLB prefetchers using academic and industrial workloads while proposing two novel concepts, ATP and SBFP, that push the envelope of hardware

TLB prefetching. However, there is still large room for improving the performance of hardware TLB prefetching since the oracle TLB prefetcher, presented in Section 3.3, improves geometric mean performance over ATP combined with SBFP by 33.3% (on average) across all benchmarks evaluated in this study.

Next, we briefly present interesting future research directions in the domain.

#### ML-based TLB Prefetching

There is a large body of works that design hardware cache prefetchers based on known ML algorithms [76, 80, 135, 182, 216, 230, 290]. However, there is no prior work examining whether ML-based TLB prefetchers could accurately prefetch for the TLB miss stream.

#### VIVT Caches and TLB Prefetching

VIVT caches remove the TLB lookup from the critical path of accessing memory, as explained in Section 2.3.3.2. The key advantage of VIVT caches is that they permit TLBs to scale to larger sizes because the TLB lookup needs to be performed after the L1 cache lookup. Consequently, sophisticated TLB prefetchers (including ML-based TLB prefetchers) can be employed alongside the different TLB levels when VIVT caches are used.

#### Storage Medium for Prefetches and Replacement Policy

This potential research direction would answer whether the PB could be entirely removed or shrink to smaller sizes by placing the TLB prefetches with high confidence directly into the TLB hierarchy. What policy should be used for driving replacements in the PB? What replacement policy should be used for the TLBs when there is no PB in the system? Should a TLB replacement policy treat differently demand TLB entries and prefetched TLB entries?

#### TLB Prefetching Under Virtualization

This potential research would quantify the benefits of applying TLB prefetching in virtualized environments since 2D page walks are much costlier than 1D page walks; 2D/1D page walks can cause up to 24/4 additional memory references to obtain a requested address translation entry.

# 4 Markov-based Instruction TLB Prefetching

## 4.1 Introduction

The effort to reduce address translation overheads has typically targeted data accesses since they constitute the overwhelming portion of the TLB misses in desktop, high performance computing (HPC), and big data applications [82, 85, 86, 165, 167, 188, 214, 219, 221, 241, 243, 300]. The address translation cost of instruction accesses has been relatively neglected due to historically small instruction footprints of applications. However, recent academic and industrial studies [166, 209, 304] demonstrate that modern server and datacenter applications feature not only large datasets, but also large code footprints owing to their huge binaries and deep software stacks. As a result, these applications place tremendous pressure on processor front-end structures (*e.g.*, L1i, iTLB), compromising performance due to unavoidable pipeline stalls. To make matters worse, the front-end performance bottleneck is likely to be exacerbated since the instruction footprint of modern server and datacenter applications is annually increasing within the 20-30% range [166], significantly outpacing the growth in the front-end structure sizes.

When it comes to instruction address translation, TLB pressure caused by massive code working set sizes is amplified by contention in the L2 TLB,[1] which is shared between instruction and data translations, as explained in Section 2.3.3.2. Instruction references evict useful data translation entries from the TLB and vice versa, imposing additional performance penalties. However, instruction TLB misses are more critical than data TLB misses since instruction references are on the critical path of pipeline execution, while data TLB misses can overlap independent instructions thanks to ILP and MLP in out-of-order cores, partially hiding their latency costs. Indeed, a recent work [209] shows that instruction TLB misses are a critical bottleneck in Facebook workloads. In addition, recent work from Google [166, 200] demonstrates that their server and datacenter workloads experience high pressure on front-end structures while highlighting that most of their applications exhibit high instruction growth rates. The main takeaway is that instruction TLB misses are a growing problem in servers and datacenters.

Surprisingly, minimal attention has been paid to the instruction address translation costs of server and datacenter applications by the research community. Existing software approaches comprise either compile-time techniques for code layout optimization [207] or operating system schemes leveraging large pages [107, 177, 304]. On the hardware side, there is no prior work specifically targeting the instruction address translation bottleneck. However, previously proposed hardware schemes, originally conceived to target data TLB misses, could also be effective for eliminating instruction TLB misses. Incremental approaches try to increase TLB reach [103, 219, 221] but they have narrow applicability to the instruction address translation problem since they are limited by coalescing opportunities exposed by the application and the OS and may also introduce new security problems. Disruptive approaches [51, 169] call for an overhaul of the virtual memory subsystem, which hinders their adoption and may bring up new security vulnerabilities. Finally, hardware TLB prefetching [165], presented in Section 2.4.1.2, constitutes a fully legacy-preserving $\mu$architectural technique that relies only on the memory access patterns of the application, is independent of the system state, and does not disrupt the virtual memory subsystem. However, (i) there is no previously proposed instruction TLB prefetcher, and (ii) the effectiveness of prior data TLB prefetchers on prefetching for the instruction TLB miss stream has never been analyzed.

[1] This chapter focuses on modern TLB hierarchies with two levels and uses TLB to refer to the L2 TLB, unless stated otherwise, similar to all previous chapters.

This work reveals that instruction address translation is an emerging performance bottleneck in server applications because the large code footprints of these applications pressure the TLB hierarchy, resulting in long-latency page walks for fetching the corresponding address translations. To support this claim, we provide the first $\mu$architectural study that (i) characterizes the TLB behavior of industrial server workloads, and (ii) provides evidence that instruction address translation is a performance bottleneck in servers. Specifically, on a suite of contemporary industrial server workloads, we find that over 40% of all TLB misses are caused by instruction references. Our findings corroborate the conclusions of previous industry works showing instruction TLB pressure to be a performance bottleneck in their workloads [166, 200, 209].

To alleviate the instruction address translation bottleneck, we focus on hardware TLB prefetching. First, we show that prior data TLB prefetchers, presented in Section 2.4.1.2, are ineffective at capturing the TLB misses because (i) they correlate patterns with features that are unable to provide accurate instruction TLB prefetches, and (ii) they use access recency for choosing prefetch candidates which does not correlate well with instruction TLB misses. When applied to instruction TLB prefetching, existing data TLB prefetchers improve the performance on industrial server workloads by up to 1.6%, whereas the opportunity from oracle instruction TLB prefetching is 11.1%. Second, our analysis on the instruction TLB miss behavior of industrial server workloads draws the following key findings: (i) instruction TLB misses follow a skewed distribution, with a modest number of virtual pages responsible for the majority of the misses, (ii) instruction TLB miss patterns are mostly irregular while having limited spatial locality restricted to a small region around the triggering miss, and (iii) the instruction TLB miss stream correlates well with the miss frequency of instruction pages.

Furthermore, we examine the state-of-the-art instruction cache prefetchers presented in the 1$^{st}$ Instruction Prefetching Championship (IPC-1) [32] and conclude that they, too, are ineffective at prefetching for the instruction TLB miss stream. Instruction prefetchers target the L1i cache and typically find the needed cache blocks in the L2C or the LLC [117, 233], which means that they are tuned for relatively short prefetch distances. Meanwhile, instruction TLB misses result in page walks that cause serialized accesses to the memory hierarchy. Depending on the memory hierarchy level where these accesses are served, the page walk can take from tens to hundreds of cycles, which cannot always be covered by instruction cache prefetchers.

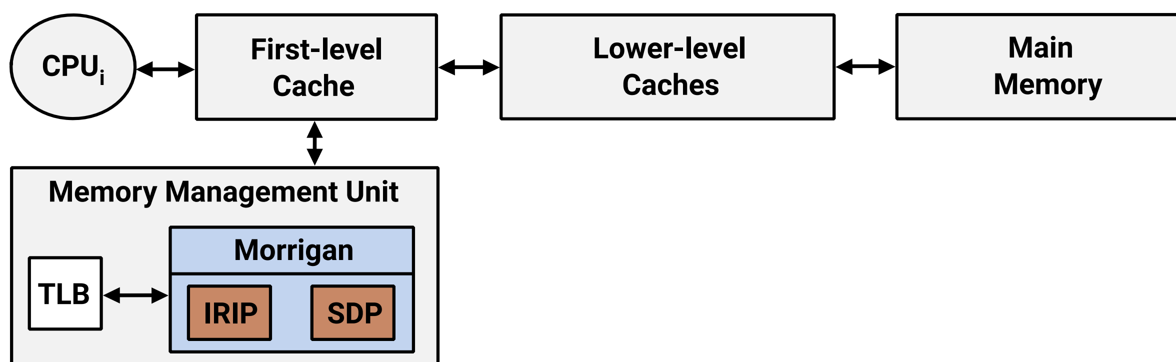


Figure 4.1: Morrigan integrated into a modern $\mu$architecture.

This chapter introduces *Morrigan*, a composite $\mu$architectural TLB prefetcher for instruction accesses whose design is inspired by our analysis findings. To the best of our knowledge, this is the first work to characterize instruction TLB misses and the first instruction TLB prefetcher. Morrigan is composed of two complementary prefetch engines. The first module is the *Irregular Instruction TLB Prefetcher (IRIP)*, an ensemble of four prediction tables that efficiently build and store variable length Markov chains from the instruction TLB miss stream. IRIP is enhanced with a new replacement policy, named *Random-Least-Frequently-Used (RLFU)*, that drives replacements based on a frequency stack of instruction TLB misses. RLFU uses randomness to avoid evicting recently installed but not yet frequently accessed entries, thus efficiently accommodating changes in the instruction access patterns, *e.g.*, due to phase-based behavior. The second module of Morrigan is the *Small Delta Prefetcher (SDP)*, a sequential prefetcher activated when the IRIP module is unable to produce new prefetches. Finally, both IRIP and SDP exploit page table locality, presented in Section 3.1, to perform cost-effective spatial prefetching at zero cost. Figure 4.1 depicts Morrigan (and its constituent prefetchers) integrated into a modern $\mu$archtecture.

In summary, this chapter makes the following contributions:

- We provide the first study on instruction TLB prefetching using a set of 45 industrial server workloads provided by Qualcomm for the 1$^{st}$ Contest on Value Prediction (CVP-1) [19]. Key conclusions of the study are that (i) state-of-the-art designs of data TLB prefetchers, presented in Section 2.4.1.2, are unable to cover instruction TLB misses, and (ii) L1i cache prefetchers are ineffective at eliminating instruction TLB misses.

- We reveal that instruction TLB misses (i) follow a skewed distribution, with a modest number of instruction pages responsible for the majority of the instruction TLB misses, and (ii) have spatial locality limited to a small region around the triggering miss.

- We propose *Morrigan*, a novel instruction TLB prefetcher composed of two specialized prefetch engines: (i) *Irregular Instruction TLB Prefetcher (IRIP)*, a novel Markov-based prefetching module that leverages a new frequency-based replacement policy, named *Random-Least-Frequently-Used (RLFU)*, to manage its internal state, and (ii) *Small Delta Prefetcher (SDP)*, an enhanced prefetcher that uses small strides for prefetching.

- Across a set of 45 contemporary industrial server workloads [19, 32], Morrigan provides a geometric mean speedup of 7.6% and reduces the references to the memory hierarchy due to demand page walks for instructions by 69% over a baseline that does not employ instruction TLB prefetching.

## 4.2 Background

All necessary background information about architectural support for virtual memory systems and hardware TLB prefetching are presented in Sections 2.3.3 and 2.4, respectively. This section builds on top of the concepts presented in Sections 2.3.3 and 2.4, thus understanding these concepts is a prerequisite for following the rest of this chapter.

This chapter focuses on x86-64 architectures which constitute the dominant processor architecture deployed in today's datacenters [68] and considers a system with a 2-level TLB hierarchy, a 4-level radix tree page table, MMU-Caches with 3 levels (called *Page Structure Caches (PSCs)* in x86-64 architectures), and a 3-level cache hierarchy, similar to Figure 2.20 in Section 2.4. In addition, it considers the most common scenario, described in Section 2.4, where TLB prefetching is solely applied for the L2 TLB, and the prefetched PTEs are placed into a dedicated TLB buffer, named Prefetch Buffer (PB). The only difference compared to Section 3.2 is that the TLB prefetcher is activated upon L2 TLB misses for instruction accesses while there is no prefetching happening upon L2 TLB misses for data references. For the rest of this chapter, we use TLB prefetcher to refer to an L2 TLB prefetcher for instruction accesses, unless stated otherwise. Finally, this chapter uses the concept of *free TLB prefetching* that exploits the locality in the last level of the radix tree page table, presented in Section 3.2.1, to perform cost-effective spatial instruction TLB prefetching.

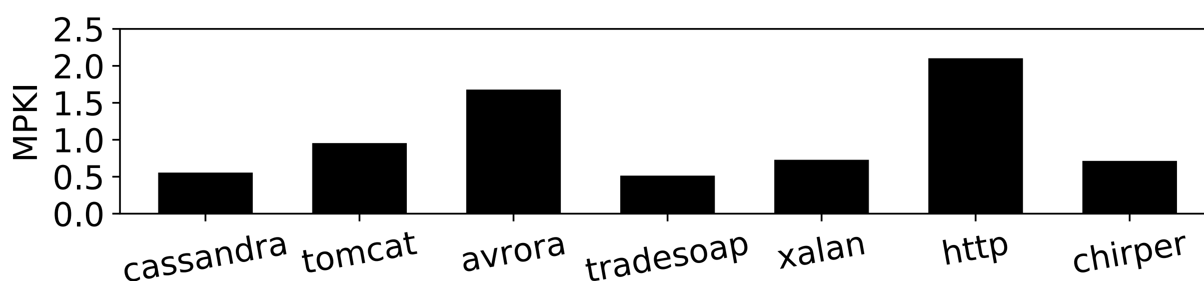


Figure 4.2: Instruction TLB MPKI of Java server workloads from the Java DaCapo [89] and Java Renaissance [224] benchmark suites.

## 4.3 Motivation

This section elaborates on the front-end bottleneck of servers and motivates the need for new approaches that alleviate the instruction address translation overheads, highlighting the potential performance gains of applying instruction TLB prefetching.

### 4.3.1 Front-end Bottleneck

Modern server workloads have massive instruction working sets that span many levels of the software stack, making the front-end of the processor a major performance pain point [117]. Indeed, recent work from Google [166, 200] demonstrates that their server workloads face severe problems due to pressure on front-end structures. Moreover, they highlight that the front-end bottleneck is increasing, since most of these server applications exhibit high instruction growth rates (∼20-30% per year), outpacing the growth in the instruction cache and TLB sizes. Specifically, Kanev *et al.* [166] reveal that the front-end stalls of the Google server workloads account for 15-30% of pipeline slots, with many workloads being starved for instructions for 5-10% of cycles. Similarly, another recent work [209] reveals that Facebook server workloads experience serious performance bottlenecks due to front-end stalls mostly caused by instruction TLB misses.

To justify that instruction address translation is a significant bottleneck in server applications, we analyze the instruction TLB behavior of server applications from the (i) Java DaCapo suite [89] (`cassandra`, `tomcat`, `avrora`, `tradesoap`, `xalan`), and (ii) Java Renaissance suite [224] (`http`, `chirper`). We run these Java server applications on an Intel Skylake CPU with a 1536-entry TLB, and gathered performance counters associated with the instruction TLB accesses using the *perf* utility [37].

Figure 4.2 presents the instruction TLB MPKI rates of these workloads; note that TLB refers to L2 TLB, as stated in Section 4.2. For this experiment, we enable the *Transparent Huge Page support* [43] to use 2MB pages for data accesses while mapping the code pages into 2MB pages using *libhugetlbfs* [46] since there is no transparent way to map code pages into large pages today; Section 4.5 elaborates on the implications of using large pages for code. We observe that, even with large pages enabled, these applications experience high instruction TLB MPKI rates that range between 0.6 and 2.1, which results in over 5% of their execution cycles spent in instruction TLB miss handling.

Intuitively, the increasing instruction footprint of server applications affects the performance of the iTLB as well as the L2 TLB, since more instruction page table entries (PTEs) must be allocated to map the instruction working set of the applications. Therefore, the iTLB experiences high MPKI rates and, as a result, more requests for instruction address translations are sent to the L2 TLB. Since the L2 TLB contains both data and instruction PTEs, there is increasing contention between them. Higher contention leads to more frequent TLB misses that must be resolved through a long-latency page walk. However, instruction TLB misses are more critical than data TLB misses because instruction references are on the critical path of execution, while data misses can overlap the execution of independent instructions in out-of-order processors. This is the reason why processor vendors (i) employ larger iTLBs than dTLBs (*e.g.,* Intel's Skylake 2018 chips have a 128-entry (8-way) iTLB and a 64-entry (4-way) dTLB [48]), and (ii) keep increasing the L2 TLB size – from a 512-entry L2 TLB for Sandy Bridge [40] to a 1024-entry L2 TLB for Haswell [28], and a 1536-entry L2 TLB for Coffee Lake [14].

### 4.3.2 Analyzing Industrial Server Workloads

To validate the observations presented in Section 4.3.1, this section analyzes the L1i cache and TLB behavior of 45 industrial server workloads provided by Qualcomm for the 1st Contest on Value Prediction (CVP-1) [19] and the 1st Instruction Prefetching Championship (IPC-1) [32]. The Qualcomm server workloads were also used in recent works on TLB management [196, 284]. We further study the SPEC CPU 2006 [41] and SPEC CPU 2017 [42] benchmark suites to demonstrate that HPC applications have tiny code footprints. For the rest of this chapter, we use QMM and SPEC to refer to the Qualcomm workloads, and the SPEC CPU 2006 and SPEC CPU 2017 workloads, respectively. This motivational analy-

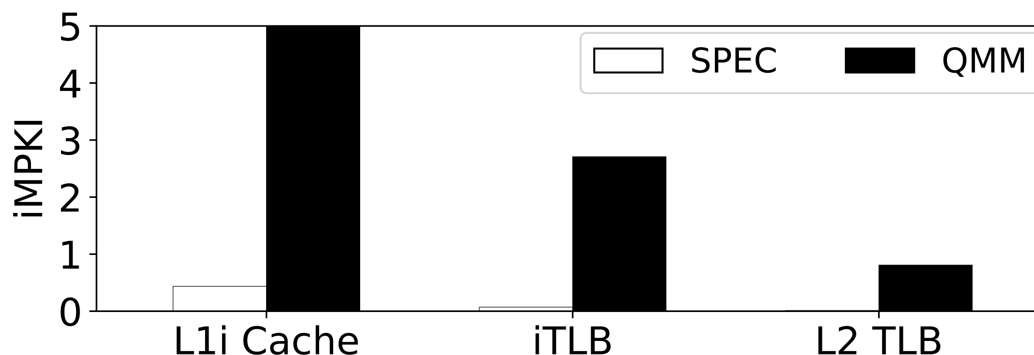


Figure 4.3: Average instruction MPKI (iMPKI) for front-end structures (L1i cache, iTLB, L2 TLB) across the QMM and SPEC suites. The L2 TLB iMPKI considers only the instruction L2 TLB misses.

sis is conducted using an enhanced version of the ChampSim simulator [13, 124], similar to Chapter 3. Finally, Section 4.5 explains in detail our simulation infrastructure, experimental setup, and the considered workloads.

Figure 4.3 presents the average MPKI rates of the L1i cache, the iTLB, and the L2 TLB (considering only the instruction misses) for the SPEC and the QMM server workloads. The main conclusions arising from Figure 4.3 are: (i) the QMM server workloads experience an order of magnitude more instruction misses in the three hardware structures (L1i, iTLB, L2 TLB) compared to the SPEC workloads, corroborating the conclusions of prior industrial works from Google [166, 200], presented in Section 4.3.1, and (ii) the instruction L2 TLB MPKI rates of the QMM server workloads are similar to the ones of the Java DaCapo and Java Renaissance workloads, presented in Section 4.3.1.

Focusing on the QMM server workloads, we measured the fraction of the L2 TLB misses that are caused by instruction and data references. We found that the instruction TLB misses constitute 41.6%, on average, of the total TLB misses (the rest 58.4% are data TLB misses). We further measured that the average page walk latency of instruction TLB misses and data TLB misses is 69 cycles and 112 cycles, respectively. Higher page walk latency is observed for the data TLB misses because the data footprint of the QMM server workloads is larger than their instruction footprint, thus, data page walks experience worse cache locality than the instruction page walks, resulting in higher page walk latency costs. However, unlike data TLB misses – whose latency can be partially hidden by exploiting ILP and MLP in out-of-order cores – instruction TLB misses cause unavoidable pipeline stalls. Therefore, instruction TLB misses constitute an important performance bottleneck in server workloads.

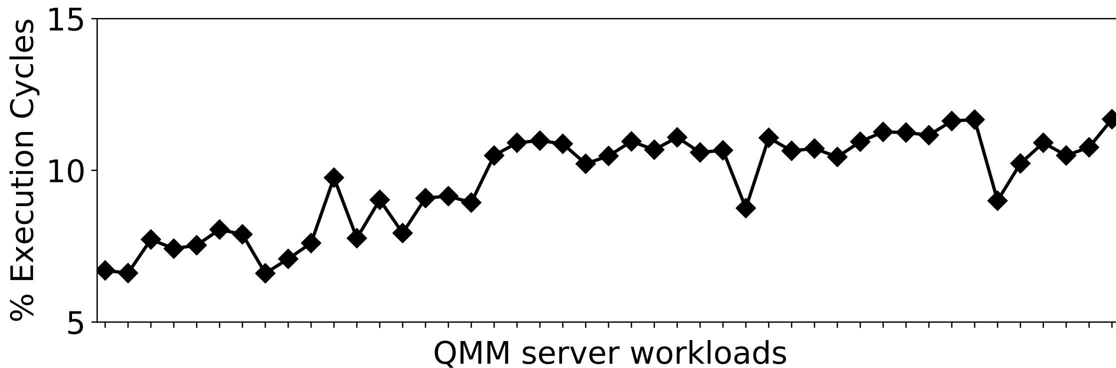


Figure 4.4: Cycles spent serving instruction TLB accesses across all QMM server workloads.

Intel's VTune profiler [21, 22, 47, 106] considers instruction address translation as a bottleneck when the stall cycles due to instruction TLB accesses represent more than 5% of the total execution cycles. Figure 4.4 shows the cycles spent serving instruction TLB accesses as a percentage of the total execution cycles for the QMM server workloads. We observe that the QMM server workloads spend 6.6%-11.7% of their total execution cycles serving instruction TLB requests, exceeding the 5% threshold. The main takeaway is that instruction address translation is a significant bottleneck for the QMM server workloads.

### 4.3.3 Understanding the Instruction TLB Misses

This section examines the behavior and the properties of the instruction TLB misses experienced by the QMM server workloads. To do so, it considers different metrics to draw conclusions capable of motivating the design of a novel instruction TLB prefetching module.

#### Delta Distribution

Figure 4.5 depicts the accumulative distribution of deltas (absolute values) between virtual pages that produce consecutive instruction TLB misses for the QMM server workloads in order of increasing deltas. While we observe a wide distribution of deltas, we note that small deltas occur frequently; deltas from 1 to 10 account for 19% of the total deltas.

| ***Finding 1.*** *Instruction TLB misses have only limited spatial locality mostly restricted to a small region around the triggering miss.* |
| --- |

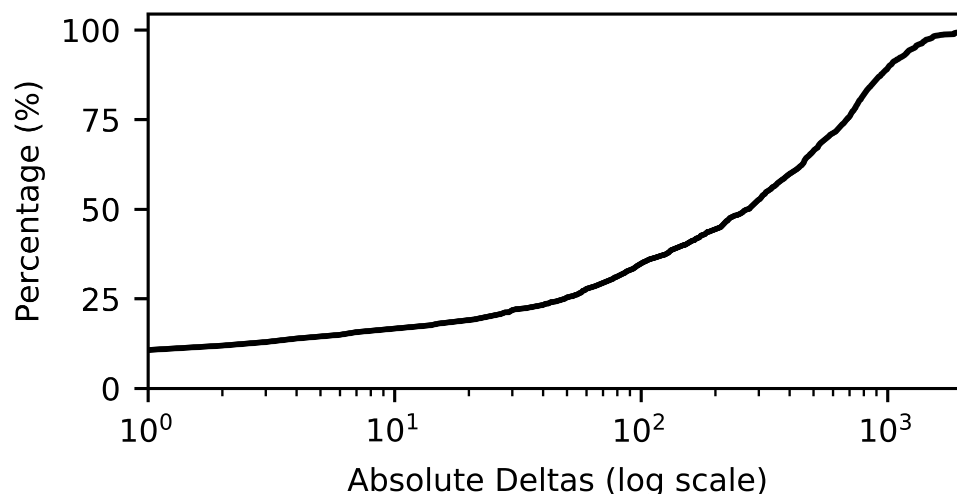


Figure 4.5: Accumulative distribution of deltas (absolute values) between pages that produce consecutive instruction TLB misses across all considered QMM server workloads.

### Distribution of Instruction TLB Misses

Figure 4.6 plots the accumulative distribution of instruction TLB misses per page in order of decreasing page occurrence frequency, considering a set of four representative QMM server workloads. The rest of the QMM server workloads follow a distribution that is either close or in between the ones presented in Figure 4.6, thus not plotted for readability. Looking at Figure 4.6, it can be observed that a small number of instruction pages is responsible for a significant fraction of all instruction TLB misses. Specifically, 400-800 instruction pages are responsible for 90% of the instruction TLB misses across all QMM server workloads.

***Finding 2.*** *Most instruction TLB misses can be attributed to a modest number of pages.*

### Identifying Chains of Instruction TLB Misses

This section is primarily focused on answering whether the instruction TLB misses of the QMM server workloads exhibit chaining behavior, *i.e.*, occur in a specific order. Secondly, it quantifies how often such chaining behavior occurs. To provide a clear answer to these questions, we define *successor page* as a page immediately following a given page in the

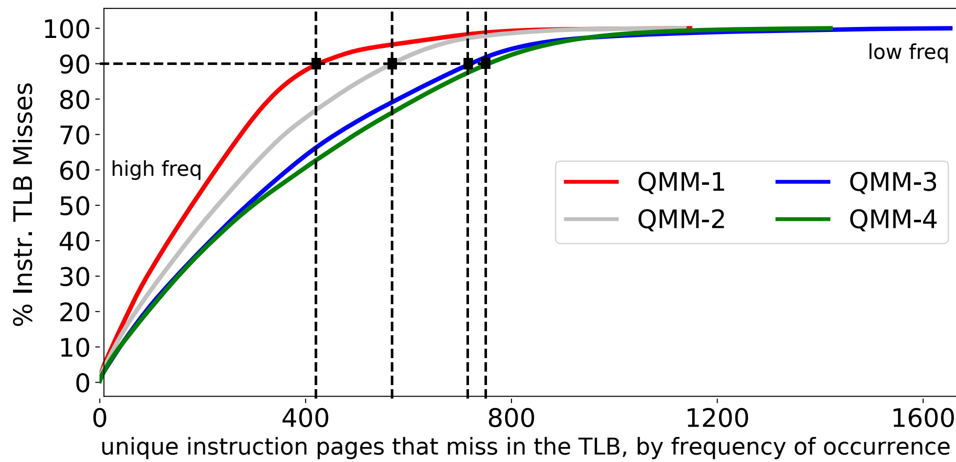


Figure 4.6: Instruction pages that produce at least one TLB miss, sorted by TLB miss frequency.

instruction TLB miss stream.[1] Figure 4.7 shows a breakdown of the average number of successors per each instruction page that missed in the TLB, across all QMM server workloads. It can be observed that (i) a significant fraction of instruction pages has only 1 or 2 successor pages, (ii) the percentage of instruction pages that have up to 4 and up to 8 successor pages is also large, and (iii) only a small number of instruction pages has more than 8 successor pages.

Figure 4.7 reveals that a big fraction of the instruction pages has more than 2 and up to 8 successors, taking into account all instruction pages that miss in the TLB. To alleviate the instruction address translation bottleneck, it is natural to mainly focus on the instruction pages that miss the most in the TLB. Figure 4.8 shows the probability of accessing a specific successor page for the top 50 instruction pages that miss the most in the TLB, across all QMM server workloads. On average, 51% of the time the most-frequent successor is accessed after an instruction TLB miss, while 21% and 11% of the time the same second and third most-frequent successors are accessed after a miss, respectively. The remaining 17% of the time, the access after an instruction TLB miss is to a less-frequent successor.

| ***Finding 3.*** *Instruction pages that miss frequently in the TLB have only a few likely successor pages whose reference probability is high.* |
|---|

[1] Page Y is a successor of page X if an instruction TLB miss on page X is immediately followed by an instruction TLB miss on page Y.

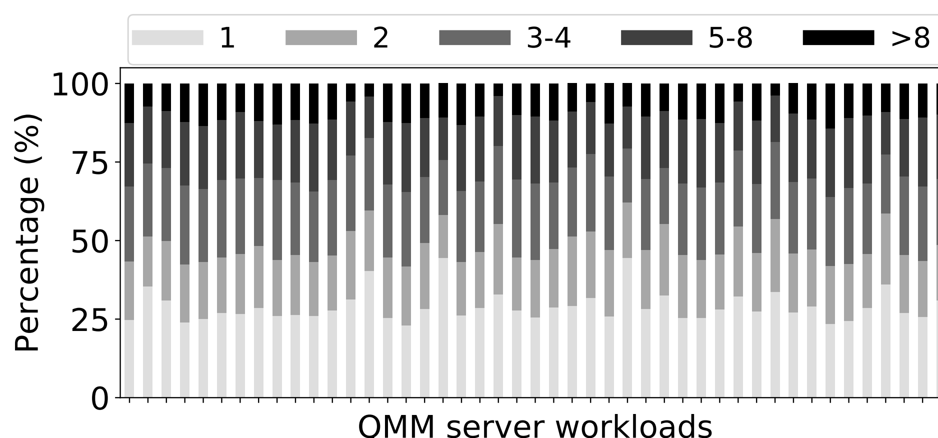


Figure 4.7: Number of successor pages per instruction page that misses in the TLB, across all considered QMM server workloads.

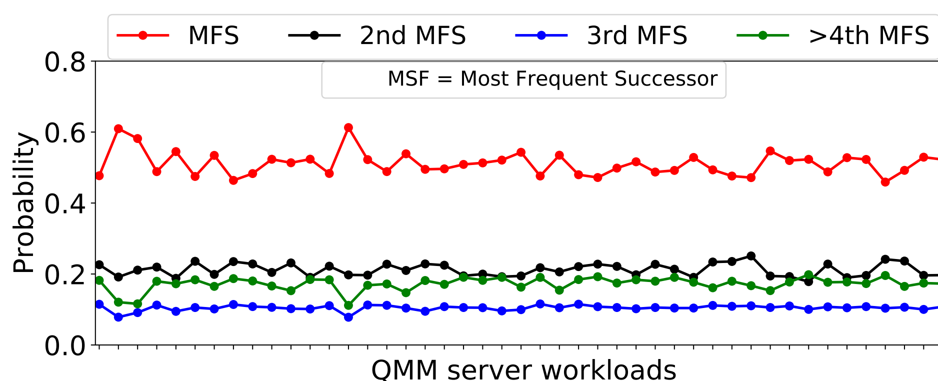


Figure 4.8: Probability of accessing the same successor page after an instruction TLB miss for a given instruction page, across all QMM server workloads.

### 4.3.4 Can Existing data TLB Prefetchers Help?

This section quantitatively answers whether previously proposed data TLB prefetchers can capture the instruction TLB miss patterns of server workloads. To do so, we measure the effectiveness of SP, ASP, DP, and MP, presented in Section 2.4.1.2, at prefetching for the instruction TLB miss stream of the QMM server workloads. For this evaluation, we set the configuration parameters of each data TLB prefetcher as proposed in the original papers; Section 4.5.2 presents their configuration. Finally, all evaluated TLB prefetchers store the prefetched PTEs into a 64-entry PB that uses the FIFO replacement policy, similar to Chapter 3, and new prefetches are issued upon instruction TLB misses, as explained in Section 4.2.

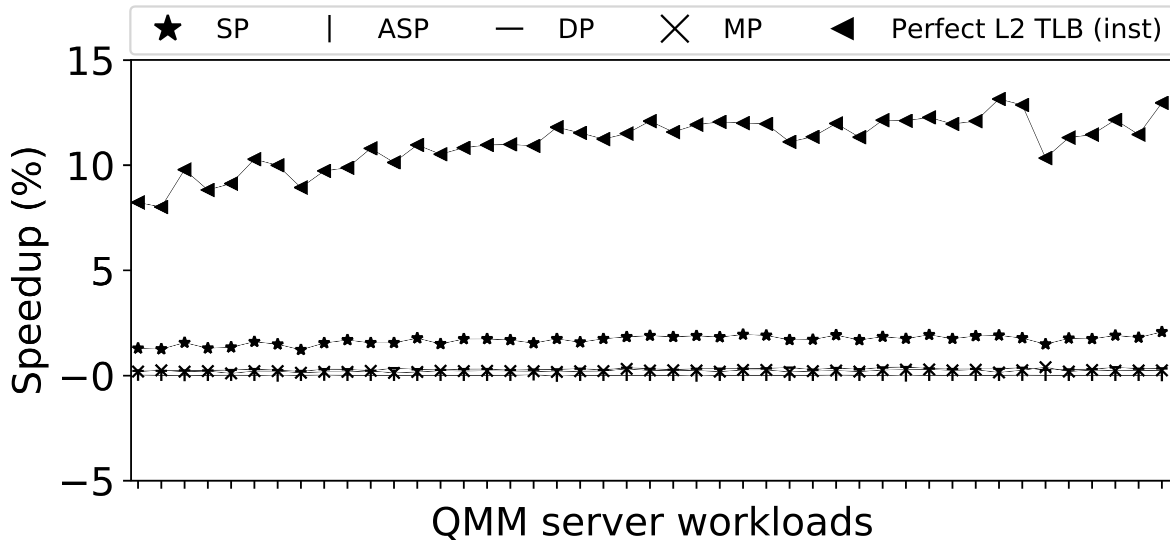


Figure 4.9: Performance comparison between the previously proposed data TLB prefetchers (SP, ASP, DP, MP) when prefetching for the instruction TLB miss stream of the QMM server workloads and the ideal scenario where all instruction TLB accesses are hits (Perfect L2 TLB (inst)). Higher is better.

Figure 4.9 illustrates the performance of the existing data TLB prefetchers (SP, DP, ASP, MP) when prefetching for the instruction TLB miss stream of the QMM server workloads, including the performance of an idealized scenario; a Perfect TLB for instruction accesses where all instruction TLB lookups are hits (Perfect L2 TLB (inst) in Figure 4.9), thus there are no page walks due to instruction references happening in the system. This ideal scenario quantifies the upper bound for performance improvement by optimizing TLB operation for instruction references. Finally, all speedups presented in Figure 4.9 are computed over a baseline that does not apply TLB prefetching.

Looking at Figure 4.9, we observe that the ideal scenario (Perfect L2 TLB (inst)) delivers the highest performance enhancements among the evaluated schemes. Specifically, Perfect L2 TLB (inst) delivers a geometric mean speedup of 11.1% across all QMM server workloads. Meanwhile, the previously proposed data TLB prefetchers provide negligible speedups because they are mainly unable to capture the instruction TLB miss patterns of the QMM server workloads. SP improves geometric mean performance by 1.6% because some of the instruction TLB miss patterns are sequential but it fails at capturing the complex delta patterns, depicted in Figure 4.5, corroborating our 1$^{st}$ analysis finding, stating that instruction TLB misses have only limited spatial locality mostly restricted to a small re-

gion around the triggering miss. On the other hand, ASP and DP provide almost no speedup over the baseline that does not apply TLB prefetching because they use features (PC and distances, respectively) that do not correlate well with the instruction TLB misses, thus, their prediction tables experience massive conflicting accesses (96.3% and 93.7%, respectively). Intuitively, we were expecting MP to provide good performance enhancements since Figure 4.7 shows that the instruction pages that miss in the TLB have a small number of successors pages. Yet we observe that MP performs poorly, improving performance by a mere 0.2% over the baseline that does not apply TLB prefetching.

To explain the poor performance of MP and examine its potential for instruction TLB prefetching, we implement and evaluate two idealized versions of MP that are not feasible for a realistic design. Both versions leverage an unbounded prediction table that accommodates all instruction pages that miss in the TLB. These idealized MP versions only differ in the number of successor pages they can store per prediction table entry; the first version maintains up to two successor pages per prediction table entry, and the other can store any number of successor pages per prediction table entry. Our explorative evaluation revealed that the unbounded MP with two and infinite successor pages per prediction table entry deliver 7.9% and 10.3% geometric mean speedups, respectively.

There are three important conclusions from this study. First, increasing the number of entries in the prediction table significantly improves MP's performance–from 0.2% geometric mean speedup with the baseline configuration that uses a 128-entry prediction table to 7.9% geometric mean speedup with an infinite number of prediction table entries. The arising conclusion is that Markov-based prefetching has the potential to reduce the instruction TLB miss rates of the QMM server workloads. Second, our analysis indicates that the replacement policy of MP is one of the reasons why MP does not improve performance with practical prediction table sizes. Since MP uses the LRU policy we conclude that recency is not a useful feature for driving replacement decisions. Finally, accommodating multiple successors per prediction table entry, beyond just two, further increases the geometric mean speedup from 7.9% to 10.3%, which approaches the performance of the idealized scenario (11.1%) where all instruction references hit in the TLB, presented in Figure 4.9.

***Finding 4.*** *A Markov prefetcher has potential for instruction TLB prefetching but it requires dynamically building variable length Markov chains out of the instruction TLB miss stream in a storage-efficient manner and an effective replacement policy for its prediction table.*

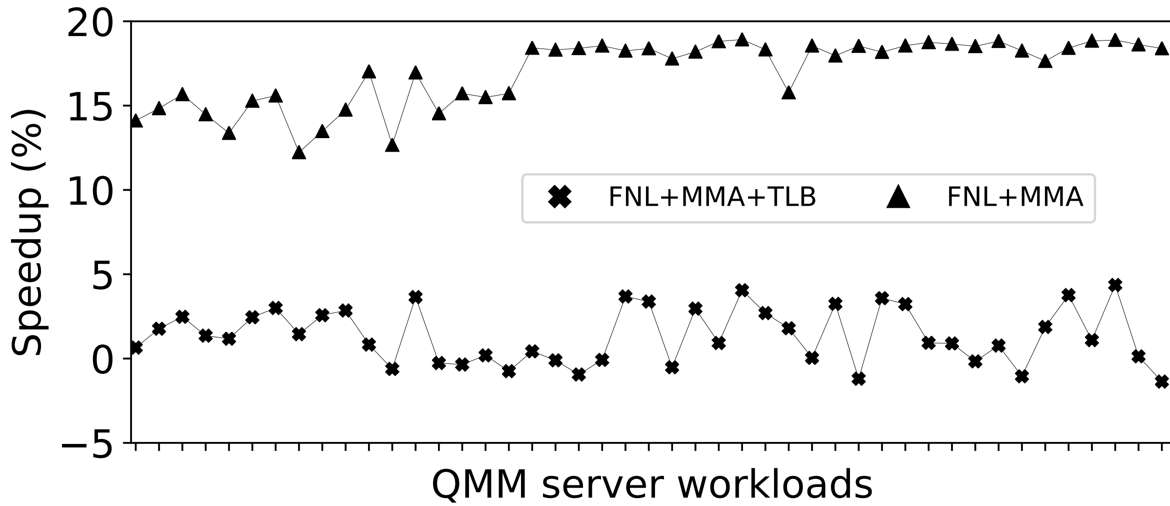


Figure 4.10: Performance of the FNL+MMA L1i cache prefetcher with and without taking into account instruction address translation, across all QMM server workloads. The baseline system utilizes the next-line L1i cache prefetcher. Higher is better.

### 4.3.5 Instruction Cache Prefetching

Modern L1i cache prefetchers are allowed to trigger instruction prefetches that cross 4KB page boundaries [32]. When that happens, if the corresponding address translation is absent in the TLB, a page walk is triggered to fetch it from the page table. Therefore, L1i cache prefetchers implicitly work as instruction TLB prefetchers; however, their effectiveness in this role has not been analyzed.

To quantify how effective state-of-the-art L1i cache prefetchers are at prefetching for the instruction TLB miss stream of the QMM server workloads, we consider the three top performers of the IPC-1 contest [32]: EPI [235], FNL+MMA [245], and D-JOLT [24]. The IPC-1 simulation infrastructure does not model instruction address translation, *i.e.*, all L1i cache prefetches that cross 4KB page boundaries are translated without cost. To consider the address translation costs when applying beyond 4KB page boundaries L1i cache prefetching, we extend the IPC-1 infrastructure to store in the TLB PB the PTEs of the pages where the blocks of the beyond-page-boundaries L1i cache prefetches reside, essentially providing instruction TLB prefetches.

Our analysis indicates that the FNL+MMA prefetcher outperforms the other IPC-1 prefetchers when address translation costs are taken into account, thus our motivational analysis

focuses on this L1i cache prefetcher. Figure 4.10 presents the performance results of different scenarios that consider the FNL+MMA prefetcher. Line FNL+MMA+TLB (FNL+MMA) shows the measured performance of the L1i cache prefetcher when instruction address translation is (is not) considered. Notably, when address translation is taken into account (FNL+MMA+TLB), we observe significantly lower speedups than the ones reported in IPC-1 contest (FNL+MMA). This performance degradation comes from the L1i cache prefetches that cross 4KB page boundaries and fail to find the corresponding translation in the TLB hierarchy, thus requiring long-latency page walks to fetch it from the page table. Such prefetches hurt the timeliness of the FNL+MMA prefetcher and delay demand TLB accesses by occupying the page table walker ports, resulting in poor performance gains. In addition, we observe only a small reduction (29.6% on average) in demand instruction TLB misses because FNL+MMA is unable to cover instruction TLB misses due to their poor timeliness in the face of long-latency page walks that require serialized memory accesses. Therefore, state-of-the-art L1i cache prefetchers require a smart instruction TLB prefetcher to effectively cross 4KB page boundaries for prefetching.

***Finding 5.*** *L1i cache prefetchers are mainly ineffective at reducing instruction TLB misses due to poor prefetching timeliness.*

### 4.3.6 Putting Everything Together

Our motivational analysis provides evidence that instruction address translation is a performance bottleneck in servers and reveals that there is potential for large performance enhancements by optimizing TLB operation for instruction references in server applications. However, the design of existing data TLB prefetchers falls short at reducing the instruction TLB misses of server applications. In addition, we demonstrate that Markov-based TLB prefetching is a promising solution to the instruction address translation bottleneck, but the reported improvements assume ideal and indefinitely large prediction tables, infeasible for a realistic design. Applying Markov-based instruction TLB prefetching in the context of a properly sized prediction table requires an adaptive and storage-efficient building of Markov chains and an effective replacement policy that drives replacement decisions without taking into account the access recency of instruction pages.

## 4.4 Morrigan

To alleviate the instruction address translation performance bottleneck of server applications, this chapter proposes *Morrigan*,[1] the first ever and the state-of-the-art $\mu$architectural TLB prefetcher for instruction references. Morrigan is a fully legacy-preserving composite prefetching scheme and does not disrupt the existing virtual memory subsystem. Morrigan is also synergistic with L1i cache prefetchers as it improves the timeliness of the L1i prefetches that cross page boundaries.

### 4.4.1 Design

Morrigan is inspired by our analysis findings regarding the reuse and the locality of the instruction TLB miss stream of industrial server applications, presented in Section 4.3, and consists of two complementary prefetching modules: the *Irregular Instruction TLB Prefetcher (IRIP)* which dynamically builds and stores Markov chains out of the instruction TLB miss stream, and the *Small Delta Prefetcher (SDP)*, an enhanced sequential prefetcher. Sections 4.4.1.1 and 4.4.1.2 present the IRIP and SDP modules, respectively. Finally, Section 4.4.2 explains in detail the operation of Morrigan.

#### 4.4.1.1 Irregular Instruction TLB Prefetcher (IRIP)

**IRIP Design**

The IRIP prefetch engine is designed as a Markov prefetching module since our analysis indicates that a Markov prefetcher has potential for instruction TLB prefetching (Finding 4, Section 4.3.4). Specifically, IRIP is an ensemble of four table-based Markov prefetchers that efficiently build and store variable length Markov chains from the instruction TLB miss stream. IRIP optimizes prefetching decisions by taking into account the variable number of successor pages, as presented in Figure 4.7), of the instruction pages that miss in the TLB.

Designing the IRIP module as a monolithic Markov prefetcher with a single prediction table and a fixed number of successors per prediction table entry, as the state-of-the-art MP does (Section 2.4.1.2), results in suboptimal performance gains since it loses prefetching opportunities, as we show in Section 4.6.3.

[1] Morrigan (also known as Morrígu) is the Irish goddess of destiny.

**Prediction Table PRT-S2**

| $VPN_i$ | $D_{i1}$ | $C_{i1}$ | $D_{i2}$ | $C_{i2}$ |
|---|---|---|---|---|
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | | | | |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | | | | |

Figure 4.11: Design of the PRT-S2 prediction table. VPN, $D_{ij}$, and $C_{ij}$ refer to virtual page number, predicted distance, and confidence counter, respectively.

The IRIP module employs four prediction tables, named PRT-S1, PRT-S2, PRT-S4, and PRT-S8, that dynamically build a store variable length Markov chains from the instruction TLB miss stream. Each prediction table entry stores up to a pre-defined number of successors; the PRT-S1, PRT-S2, PRT-S4, and PRT-S8 prediction tables accommodate instruction pages that have one, two, up to four, and up to eight successor pages, respectively. In other words, each entry of the prediction table PRT-S$i$ can accommodate up to $i$ successor pages, where $i \in \{1, 2, 4, 8\}$.

Each prediction table (PRT-S1, PRT-S2, PRT-S4, PRT-S8) is realized as a set-associative buffer and stores the virtual page number (VPN) of the missed instruction for indexing, $s$ prediction slots, and $s$ confidence counters, one per prediction slot. For example, each PRT-S2 entry has $s$=2 prediction slots and $s$=2 confidence counters. The only difference in the design of the prediction tables is the number of prediction slots and confidence counters. For this reason, Figure 4.11 presents the basic design of the PRT-S2 prediction table.

A naive PRT-S2 design would store the full VPN in each prediction slot (as the state-of-the-art MP [165] does). However, such a design choice is expensive, storage-wise, since each VPN requires 36 bits of state. To lower this storage cost, IRIP stores into the prediction slots the distances between the current and the previous virtual pages that produced an instruction TLB miss. This approach lowers the amount of storage for the prediction tables without any performance loss.

The confidence counters ($C_{ij}$ in Figure 4.11) associated with the prediction slots are exploited in a two-fold manner: (i) to drive the replacement policy of the prediction slots, *i.e.*, when all the prediction slots are occupied and a new distance has to be placed in one of these slots, the distance with the lowest confidence is replaced, and (ii) the distance with the highest confidence is selected to apply spatial prefetching, leveraging the locality in the last level of the radix tree page table, presented in Section 3.2.1. Specifically, on prediction table hits (*e.g.*, PRT-S2 hits in Figure 4.11), IRIP issues one prefetch request per predicted distance of the hit entry. Each prefetch requires a prefetch page walk to fetch the corresponding translation from the page table, as explained in Section 2.4.1.1. At the end of a prefetch page walk, page table locality can be exploited to prefetch for free the PTEs that share the cache line with the target PTE. However, prefetching all the free PTEs in all prefetch page walks might harm performance by fetching a lot of inaccurate prefetches. To mitigate this problem, IRIP prefetches cache-line adjacent PTEs only for the distance with the highest confidence.

**IRIP Operation**

All IRIP prediction tables operate the same since the only difference in their design is the number of prediction slots and confidence counters. Therefore, we focus on a single prediction table to explain its operation. Figure 4.12 illustrates the operation of the IRIP module on PRT-S2 hits coupled with an example that assumes an instruction TLB miss for virtual page 0xA1. Initially, a PRT-S2 lookup takes place to determine if there is an entry corresponding to virtual page 0xA1 **1**. In the example, the PRT-S2 lookup experiences a hit. Consequently, the predicted distances of the hit entry (17 and 2) are separately summed with the currently missed page (0xA1) to generate new prefetch requests for pages 0xB2 and 0xA3, respectively **2**. In parallel, IRIP finds the predicted distance with the highest confidence counter **3**. Since distance 2 has the highest confidence value, IRIP applies spatial prefetching by leveraging page table locality for the prefetch 0xA3 (0xA1 + 2) **4**. Specifically, at the end of the prefetch page walk for 0xA3, IRIP leverages page table locality to also prefetch the PTEs adjacent to the PTE of 0xA3 **5**. The next step updates the content of the PRT-S2 table for future reuse. To do so, IRIP calculates the distance between the currently missed (0xA1) and previously missed (0xB5) virtual pages and stores the outcome into a register **6**. Meanwhile, IRIP finds which of the predicted distances for the previously

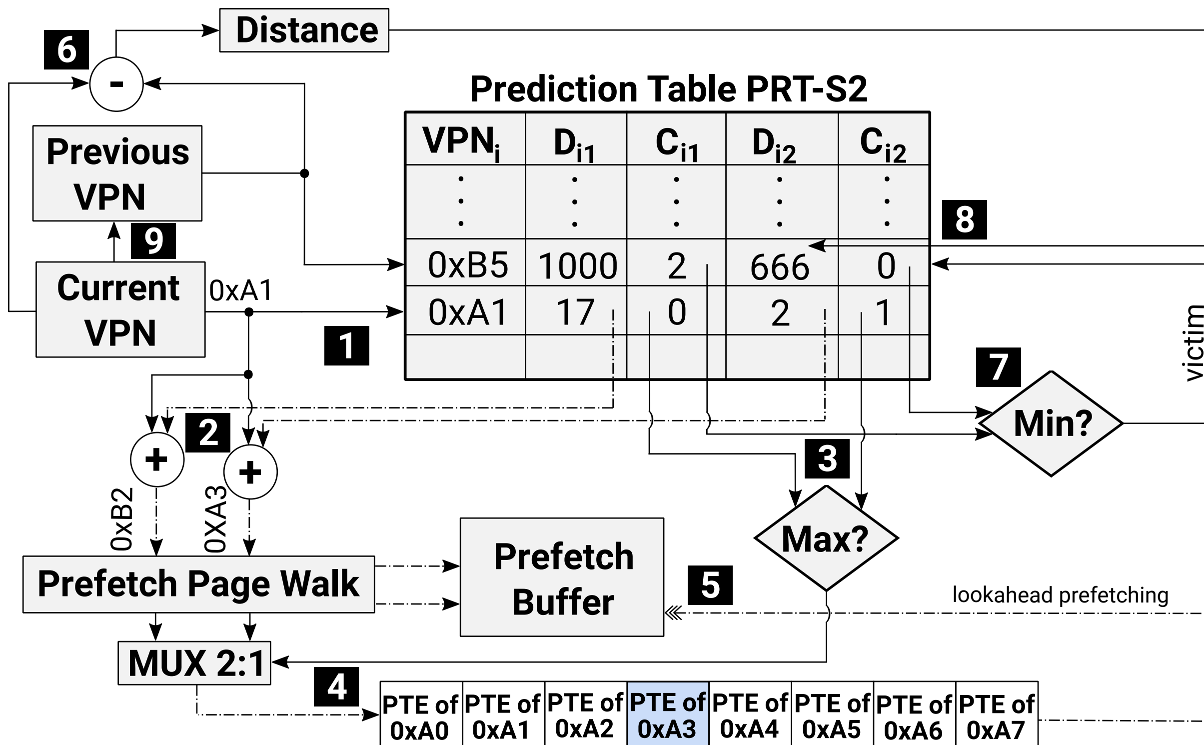


Figure 4.12: Operation of the IRIP module on PRT-S2 hits. VPN, $D_{ij}$, and $C_{ij}$ refer to virtual page number, predicted distance, and confidence counter, respectively. Diamonds indicate decision points.

missed virtual page has the lowest confidence counter 7. In this example, distance 666 has the lowest confidence, thus IRIP replaces it with the current distance while resetting the corresponding confidence counter 8. Finally, IRIP stores the currently missed virtual page into the register holding the previously missed virtual page to be used on the next IRIP operational cycle 9.

When Morrigan operates, steps 7 and 8 take place only for the PRT-S8 table; for PRT-S1, PRT-S2, and PRT-S4 Morrigan transfers the entry coupled with the new predicted distance into a prediction table with more prediction slots per entry, to avoid loosing already captured patterns. Section 4.4.2 explains in detail the operation of Morrigan, highlighting the transferring of entries between the prediction tables of the IRIP module.

*Updating the Confidence Counters.* When a prefetch is proved to be accurate, *i.e.*, it produces a hit that eliminates a demand page walk and the confidence counter of the corresponding prediction slot is incremented by 1, increasing the confidence for this prediction slot.

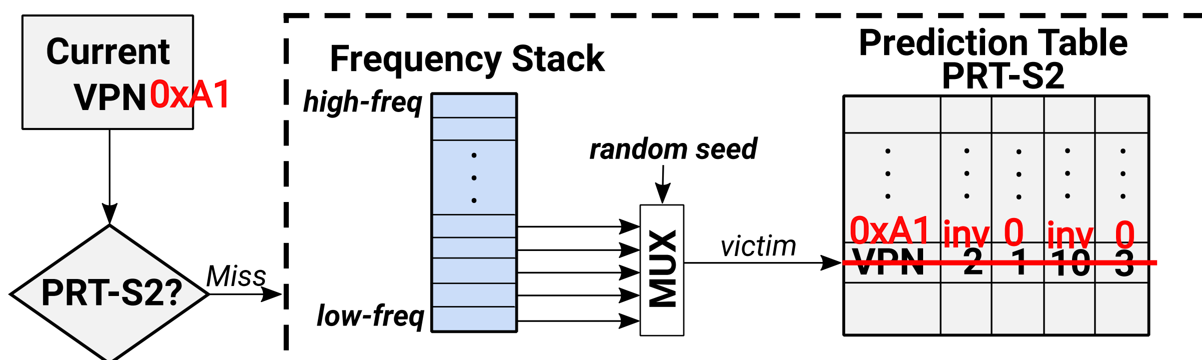


Figure 4.13: Operation of the IRIP module on PRT-S2 misses. Diamonds indicate decision points.

#### Replacement Policy

A critical aspect of the IRIP design is the replacement policy of the prediction tables. While previously proposed table-based TLB prefetchers, like MP [165], use the LRU replacement policy, our motivational analysis, presented in Section 4.3.4, reveals that LRU does not keep the most useful entries in the prediction tables because it is prone to lose track of important entries. In addition, our analysis findings indicate that the miss frequency of virtual pages is a good feature to correlate the instruction TLB miss stream. Therefore, we employ a frequency-based replacement policy for all the prediction tables of the IRIP module which (i) maintains a frequency stack of the instruction TLB misses to drive the replacement of entries on prediction table conflicts, similar to the Least-Frequently-Used (LFU) policy, and (ii) uses a random component that gives recently installed entries, which have not yet accumulated a large number of hits, a chance to persist when a replacement candidate is selected since it randomly selects for eviction one of the least frequently used entries, not necessarily the least frequently used one. This policy gives IRIP the ability to adjust to phase-based behavior in workloads. We refer to this policy as *Random-Least-Frequently-Used (RLFU)*. Finally, the complexity of RLFU is similar to the LRU policy.

Figure 4.13 depicts the operation of IRIP on PRT-S2 misses with an example that assumes a fully populated PRT-S2 and an instruction TLB miss for virtual page 0xA1 which is not stored into the PRT-S2. Consequently, IRIP needs to victimize a PRT-S2 entry to keep track of the currently missed virtual page (0xA1). In practice, one of the five least frequently accessed entries is randomly selected for replacement. The currently missed instruction page is stored in the victim's entry with cleared prediction slots and confidence counters.

A problem with a frequency-based replacement policy is that it may be slow to adapt to phase changes in application behavior (*e.g.*, when a page causes frequent instruction TLB misses in one phase but not in another). To avoid the associated performance pathologies, Morrigan periodically resets the frequency stack to better identify instruction pages causing the most instruction TLB misses in a given interval.

#### 4.4.1.2 Small Delta Prefetcher (SDP)

The *Small Delta Prefetcher (SDP)* prefetches the PTE of the virtual page adjacent to the missed virtual page, similar to SP, presented in Section 2.4.1.2. The only difference between SP and SDP is that the latter exploits page table locality, presented in Section 3.2.1, to prefetch all the adjacent PTEs within the target cache line. In this way, SDP captures the majority of the small-strided instruction TLB miss patterns, effectively addressing our first analysis finding (Finding 1 in Section 4.3.3).

To reveal the prefetching capabilities of SDP, we assume an instruction TLB miss for virtual page 0xA7. First, SDP issues a prefetch request for page 0xA8 (0xA7+1). After the completion of the prefetch page walk for page 0xA8, SDP prefetches all the PTEs that share the cache line with the PTE of page 0xA8. Note that in this example, fetching the PTEs of pages 0xA7 and 0xA8 requires two separate page walks since the PTE of 0xA7 resides in the last position of a cache line (0xA7 & 0x07) while the PTE of 0xA8 is stored in the first position within another cache line.

### 4.4.2 Operation of Morrigan

This section explains in detail the operation of Morrigan, considering the most common scenario where the instruction TLB prefetcher is invoked on instruction TLB misses, and the prefetched PTEs are stored into a Prefetch Buffer (PB), as described in Section 4.2.

Figure 4.14 illustrates step-by-step the operation of Morrigan. When an instruction TLB miss occurs **1**, the requested translation is looked up in the PB **2**. On PB misses, a demand page walk is initiated **3** to fetch the corresponding translation from the page table and store it into the TLB **4**. On PB hits, the demand page walk is avoided, the corresponding address translation entry is transferred from the PB to the TLB **5**, and in the background, we increment the confidence counter of the prediction table entry that produced the PB hit if the prefetch was produced by the IRIP module **6**.

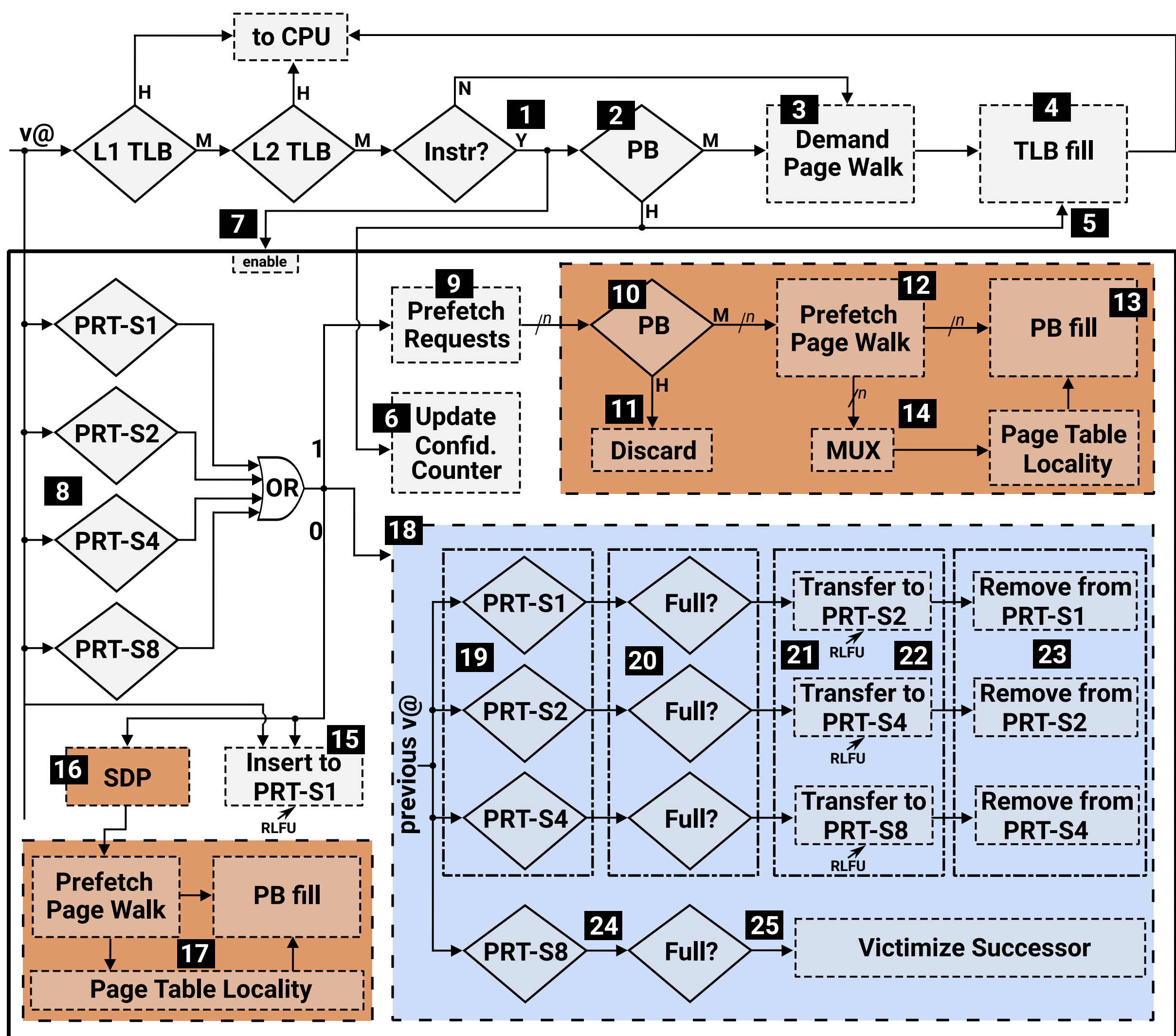


Figure 4.14: Complete operation of Morrigan in steps. Diamonds indicate decision points while dotted lines indicate actions.

Morrigan is engaged in case of either PB hit or miss 7. First, Morrigan looks up in parallel all prediction tables (PRT-S1, PRT-S2, PRT-S4, PRT-S8) of the IRIP module 8 (step 1 in Figure 4.12). When there is a hit in one prediction table,[1] Morrigan generates one prefetch request per valid prediction slot of the hit entry 9 (step 2 in Figure 4.12) of the corresponding prediction table. Before issuing new prefetch requests, Morrigan checks whether the translations already reside in the PB 10. For the prefetches that are already

[1]There is no duplication of entries in the prediction tables of the IRIP module, thus only one prediction table hit might occur per activation of Morrigan.

stored in the PB the corresponding prefetch requests are discarded 11. For the rest, separate prefetch page walks are initiated to fetch the address translation entries from the page table 12. At the end of the prefetch page walks, the corresponding PTEs are stored in the PB 13. Then, Morrigan leverages page table locality to apply lookahead prefetching with depth 1 by fetching in the PB the adjacent PTEs that are transferred together with the prefetched PTE into the cache hierarchy solely for the prefetch with the highest confidence 14 (steps 3-5 in Figure 4.12). When all the prediction tables of the IRIP module experience a miss, Morrigan needs to store the currently missed virtual page in one of the IRIP's prediction tables. Since this page does not have any valid prediction, it is always placed in the PRT-S1 table 15 but it might be moved into another prediction table if future instruction TLB misses reveal that this page has multiple successors. If PRT-S1 is full, Morrigan uses the RLFU policy to find a victim entry. Therefore, on prediction table misses Morrigan is unable to produce prefetch requests based on the IRIP module. At this point, SDP is activated and issues prefetch requests 16 by exploiting page table locality, which are eventually stored into the PB 17. SDP is enabled only when all IRIP prediction tables experience a miss because SDP has lower prefetching scope and accuracy than IRIP. Following this strategy, Morrigan does not lose any potential for performance improvement since it produces new prefetch requests on every instruction TLB miss.

In case of either hit or miss in the prediction tables of the IRIP module, Morrigan inserts the new predicted distance in one of the prediction slots of the IRIP prediction table entry that accommodates the previously missed virtual page 18. If the previously missed page resides in one of PRT-S1, PRT-S2, or PRT-S4 tables 19 and its prediction slots are fully occupied 20, then instead of victimizing one of its prediction slots, Morrigan simply transfers this entry into the next IRIP prediction table that has more prediction slots 21; if it is full, Morrigan uses the RLFU replacement policy to open up space for the transferred entry 22. Next, this entry is removed from the previous prediction table 23 to ensure that there is no duplication of entries in the prediction tables. Notably, if the previously missed instruction page resides in the PRT-S8 table and the prediction slots are fully occupied 24, the new distance is placed into the prediction slot that has the lowest confidence counter 25. Stress that in step 19 we do not search all IRIP prediction tables to find the previously missed instruction page, but we use a register to store the identifier of the table that stores the previously missed instruction page.

### 4.4.3 Additional Aspects

#### Operation on SMT Cores

Morrigan can operate under SMT colocation by sharing the IRIP module among the threads. To do so, it only requires a different register per thread holding the virtual page number that produced the previous instruction TLB miss (step 9, Figure 4.12) to ensure that each thread builds its own Markov chains without intermixing. Finally, the SDP module does not need any modification to operate under SMT colocation since it is a stateless prefetcher that does not need to differentiate among threads.

#### Synergy with L1i Cache Prefetching

Morrigan is complementary to L1i cache prefetchers because it prefetches instruction PTEs, thus improving the timeliness of L1i cache prefetches that go beyond page boundaries by avoiding long-latency page walks (Section 4.3.5). Section 4.6.5 quantifies the performance gains of using Morrigan to improve the timeliness of a state-of-the-art L1i cache prefetcher.

#### Multiple Page Sizes

Sections 4.4.1 and 4.4.2 focus on a single page size to describe the design and operation of Morrigan. This is not a limitation of the design as multiple page sizes are supported without any modification. The page size is known only after address translation, thus, Morrigan can issue two prefetches per request to target 4KB and 2MB pages. Once the page size is known, Morrigan discards the outcome of the prefetch page walk for the mismatched page size. This approach does not add complexity to the design since modern architectures support speculative page walks [215].

#### Page Replacement Policy and TLB Shootdowns

Morrigan sets the access bit of all prefetched pages since the x86 memory consistency model dictates that all TLB prefetches are obliged to do so [48]. Therefore, Morrigan does not complicate TLB shootdowns [53, 57, 88, 181, 288] because the information about the prefetched instruction PTEs is conveyed to the OS as usual. Regarding the impact on the page replacement policy, a prefetch is harmful to the page replacement policy if it is evicted

from the TLB PB without providing any hit and does not belong to the active footprint of the application. Morrigan issues prefetches based on the control-flow behavior and does not permit faulting prefetches, thus the probability of negatively affecting the page replacement policy is negligible. To annihilate this probability, Morrigan could issue a correcting page walk to reset the access bit of the PTEs that are evicted from the PB without providing any hit. These correcting page walks could be issued when the TLB MSHR is not full to avoid delaying any other page walk, as explained in Section 3.8.4.

#### Context Switches

The prediction tables of the IRIP module must be flushed upon a context switch. Their small sizes ensure that, following a context switch, they are quickly refilled. SDP is stateless; as such, it requires no action on a context switch.

#### Different Architectures

This thesis focuses on x86 architectures; however, architectural support for virtual memory used in x86 architectures [5, 30] is similar to other architectures (*e.g.*, ARM [9] and RISC-V [38]). The main takeaway is that Morrigan would be applicable to these architectures.

#### Page Table Designs

Morrigan is compatible with any multi-level radix tree page table design. Previous sections mainly focus on 4-level radix tree page tables but Morrigan is also compatible with 5-level radix tree page tables [1], and may deliver higher performance benefits because the extra page table level might increase the page walk latency. Alternatively, if a hashed page table [118, 140, 259, 300] is employed, Morrigan would operate the same since hashed page tables preserve page table locality.

#### TLB Prefetching Strategy

TLB prefetching schemes are typically engaged on TLB misses and store the prefetched PTEs into a PB, as explained in Section 2.4.1.1. Our analysis indicates that these two strategies have a positive effect on performance. Nonetheless, Morrigan could be also activated on TLB hits and store prefetched PTEs directly into the TLB.

| Component | Description |
|---|---|
| **ROB** | 352-entry |
| **L1 iTLB** | 128-entry, 8-way, 1cc, 4-entry MSHR, LRU |
| **L1 dTLB** | 64-entry, 4-way, 1cc, 4-entry MSHR, LRU |
| **L2 TLB** | 1536-entry, 6-way, 8cc, 4-entry MSHR, LRU, 1 page walk / cycle |
| **Page Structure Caches (PSCs)** | 3-level Split PSC, 2cc. PML4: 2-entry, fully assoc; PDP: 4-entry, fully assoc; PD: 32-entry, 4-way |
| **L1 iCache** | 32KB, 8-way, 4cc, 8-entry MSHR, LRU, next line prefetcher |
| **L1 dCache** | 32KB, 8-way, 4cc, 8-entry MSHR, LRU, next line prefetcher |
| **L2 Cache** | 512KB, 8-way, 8cc, 32-entry MSHR, LRU, SPP prefetcher [172] |
| **LLC** | 2MB, 16-way, 10cc, 64-entry MSHR, LRU |
| **DRAM** | 4GB DDR4, tRP=tRCD=tCAS=12, 12.8 GB/s |
| **Branch Predictor** | hashed perceptron [273] |

Table 4.1: System simulation parameters.

## 4.5 Methodology

### 4.5.1 Simulation Infrastructure

To evaluate Morrigan, we use ChampSim [13, 124], a detailed trace-based simulator that models a 4-wide out-of-order processor. We extend ChampSim to simulate a realistic x86 page table walker, modeling the variable latency cost of page walks and also the variable number of memory references they require to complete, similar to Section 3.7.1. Specifically, we added a 4-level page table, a page table walker, and a 3-level split MMU-Caches. The page table walker supports up to 4 TLB misses, similar to Skylake $\mu$architecture [48], while one page walk can be initiated per cycle. Regarding the cache hierarchy, we simulate 3 cache levels. Finally, our baseline uses the next-line L1i cache prefetcher but we also consider additional L1i cache prefetchers from IPC-1 [32] in Sections 4.3.5 and 4.6.5. Table 4.1 summarizes our experimental setup.

**SMT Colocation**

We also extended ChampSim to simulate a dual-threaded SMT core to evaluate our proposal under workload colocation. Every cycle, a different thread fetches one basic block of instructions. Note that our SMT model fully accounts for the contention due to colocation in all shared $\mu$architectural structures (TLBs, PSCs, cache hierarchy).

#### 4.5.1.1 Simulated Page Sizes

This work focuses on 4KB pages, similar to prior work [196]. So why not use large pages to mitigate the address translation overhead? Although profitable when the application exhibits high locality and the system is not fragmented, large pages are not a stop-gap solution to the address translation bottleneck for both data and code accesses. In practice, using large pages for data and code potentially hurts performance in datacenters and exposes security vulnerabilities, as we explain below. Furthermore, the performance of legacy systems and cloud applications that continue to use 4KB pages still matters for their users.

Large pages have been shown to introduce performance pathologies, particularly for servers [56, 177, 304]. Another problem is the lack of flexibility in memory management with large pages compared to standard 4KB pages [138, 141, 222]. Specifically, large pages require memory contiguity and defragmentation that is not guaranteed in datacenters due to high uptimes and the fact that datacenters handle thousands of diverse applications [141, 166, 298]. Indeed, [177] shows that memory defragmentation can result in tail latency spikes and performance variability, both of which might negatively impact the performance of datacenter applications. In addition, a recent work [141] shows that transparent 2MB support for data pages is not adequate anymore and there is a need for creating transparent support for 1GB pages. Finally, [122] reveals that large pages can harm the performance of NUMA machines; this problem might be amplified with the advent of heterogeneous memories where the OS has to migrate data between fast and slow tiers of memory.

In addition to the above, concurrently supporting multiple page sizes is a complex problem; this is the reason why Linux has support for transparent 2MB pages only for data, which, in fact, took a long time to be properly implemented [48]. Today, Linux does not have support for 2MB transparent large pages for code blocks. The only way to map executable files onto large pages in Linux is to use *libhugetlbfs* [46]. However, *libhugetlbfs* does not provide automatic and transparent support for huge page code mappings since

it requires shaping the text layout in the application's address space [108]. Indeed, a recent work [207] reveals that (i) mapping the *.text* section of server applications onto large pages provides performance degradation since it puts pressure on the limited number of L1 I-TLB entries that can accommodate large pages, and (ii) mapping too many large pages using *libhugetlbfs* in production machines makes the Linux kernel misbehave as it becomes overwhelmed by the need to relocate physical pages to satisfy requests for large pages.

Another concern with mapping code to large pages is that doing so represents a security risk. Modern systems use Address Space Layout Randomization (ASLR) to obstruct certain security attacks by making it difficult for an adversary to predict target addresses. Prior work has shown that using large pages for code significantly diminishes the effectiveness of ASLR [72, 73, 113, 249]. Another security risk is the *iTLB multihit* [34] vulnerability that arises when large pages are used for code. Specifically, when an instruction fetch hits multiple entries in the iTLB it may incur a machine check error. To mitigate this issue, cloud providers such as Microsoft Azure and Amazon force all executable instruction pages to be mapped into 4KB pages [2, 3, 4, 11, 12], removing the possibility of multiple hits.

For these reasons, we focus our evaluation on standard 4KB pages but Morrigan is entirely compatible with larger page sizes, as explained in Section 4.4.2.

### 4.5.2 Evaluated TLB Prefetchers

We implement and evaluate the previously proposed data TLB prefetchers SP, DP, ASP, and MP, described in Section 2.4.1.2 as well as our proposal, Morrigan. Table 4.2 presents their configuration parameters. TLB prefetches are placed into a dedicated TLB buffer named Prefetch Buffer (PB); Table 4.2 presents its configuration. Regarding the size of the PB that stores the prefetched PTEs, we evaluate different configurations, as shown in Table 4.2.

### 4.5.3 Workloads

Our evaluation considers a set of server workloads provided by Qualcomm for the 1$^{st}$ Contest on Value Prediction (CVP-1) [19] and the 1$^{st}$ Instruction Prefetching Championship (IPC-1) [32]. These Qualcomm server workloads were previously used in other TLB-related research works [196, 284]. For the rest of this chapter, we use QMM to refer to the Qualcomm server workloads.

| Prefetcher / Component | Description |
|---|---|
| **SP** | – |
| **DP** | Prediction Table: 64-entry, 4-way, LRU policy |
| **ASP** | Prediction Table: 64-entry, 4-way, LRU policy |
| **MP** | Prediction Table: 128-entry, 4-way, LRU policy |
| **Morrigan** | PRT-S1/S2/S4: 128-entry, 32-way, 2-bit counters, RLFU policy,<br>PRT-S8: 64-entry, 16-way, 2-bit counter, RLFU policy |
| **PB** | (16-64)-entry, fully assoc, FIFO policy |

Table 4.2: Configuration parameters of the previously proposed TLB prefetchers (SP, DP, ASP, MP), Morrigan, and the Prefetch Buffer (PB) used for storing the prefetched PTEs. The parameters for Morrigan have been empirically selected after sensitivity analysis.

Workloads with an instruction TLB MPKI of at least 0.5 are considered instruction TLB intensive, thus taken into account in our experimental campaign. In total, our evaluation considers 45 instruction TLB intensive QMM server workloads. Our simulations use 50 million warmup instructions, then 100 million instructions are executed to measure the experimental results, similar to prior work [196].

We also analyze the SPEC CPU 2006 [41] and SPEC CPU 2017 [42] benchmark suites, but we find that these workloads have an instruction TLB MPKI of 0.5 or less, so they are not considered in our evaluation. However, we use the SPEC CPU 2006 and SPEC CPU 2017 benchmark suites in Section 4.3 to prove that the findings of our work are consistent with the conclusions of previous works [166, 200].

### Colocated Workloads

Datacenters colocate applications on SMT cores for better CPU and memory utilization [166, 286]. To quantify the impact of instruction TLB prefetching under SMT colocation, we simulate a dual-threaded SMT core executing two different QMM server workloads, as explained in Section 4.5.1. Our experimental campaign, presented in Section 4.6.6, considers 50 randomly chosen pairs of QMM server workloads.

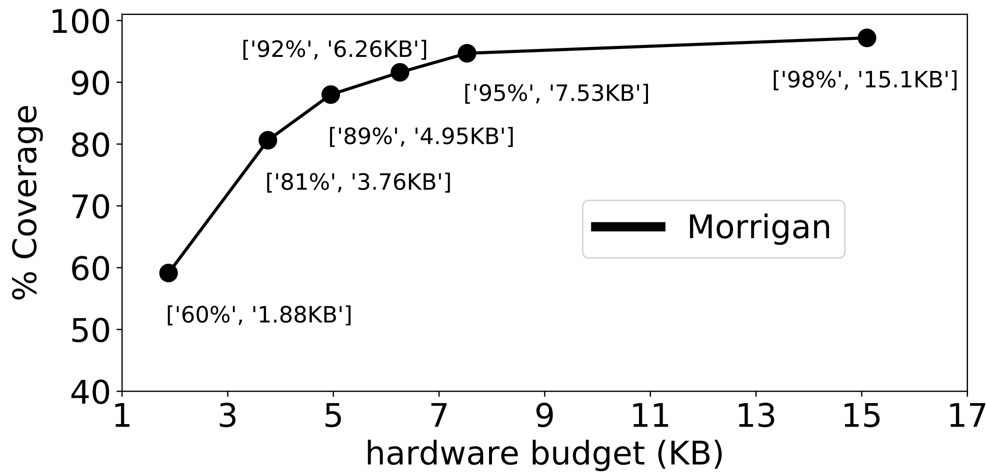


Figure 4.15: Miss coverage of Morrigan for various storage budgets. Higher is better.

## 4.6 Experimental Campaign

This section presents our experimental campaign. Section 4.6.1 focuses on the IRIP module. Section 4.6.1.2 highlights the benefits of the RLFU replacement policy. Sections 4.6.2 and 4.6.4 quantify the performance gains of Morrigan over the state-of-the-art data TLB prefetchers and other techniques that improve TLB performance, respectively. Section 4.6.3 evaluates alternative IRIP designs. Finally, Section 4.6.6 evaluates Morrigan under SMT colocation. Note that all the evaluated TLB prefetchers use a PB to store prefetched PTEs.

### 4.6.1 IRIP Module

The IRIP module of Morrigan is an ensemble of four table-based hardware Markov prefetchers. Therefore, the effectiveness of Morrigan directly depends on the number of entries in the prediction tables (PRT-S1, PRT-S2, PRT-S4, PRT-S8) of the IRIP module. Each prediction table entry requires 16 bits for storing a partial tag of the virtual page for indexing, 15 bits per predicted distance of the prediction slots, and a 2-bit saturating counter per predicted distance, as explained in Section 4.4.1.1 and Table 4.2.[1] Subsequent sections examine the impact of different parameters on the effectiveness of the IRIP module. Note that Sections 4.6.1.1 and 4.6.1.2 consider fully associative prediction tables and a 64-entry PB; Section 4.6.1.3 examines the impact of different prediction table associativities and PB sizes.

[1]The number of bits for the partial tags, predicted distances, and saturating counters were chosen empirically.

#### 4.6.1.1 Miss Coverage

This section examines the impact of different storage budgets on the miss coverage of Morrigan. To do so, Figure 4.15 presents the miss coverage of Morrigan across all the QMM server workloads as a function of different storage budgets. Starting with small storage budgets, we observe a large increase in the miss coverage of Morrigan as the storage budget increases. However, after 5KBs of storage, the miss coverage begins to plateau. Finally, we observe that going beyond 7.5KB of storage budget provides negligible benefits.

#### 4.6.1.2 Replacement Policy

The prediction tables of the IRIP module leverage the RLFU replacement policy, presented in Section 4.4.1.1. To highlight the benefits of the RLFU policy we compare it against the following alternatives: (i) LRU policy, (ii) Random policy, and (iii) LFU policy that replaces the least frequently accessed entry. Figure 4.16 depicts the miss coverage of Morrigan when the IRIP module leverages the above explained replacement policies as a function of different budgets, similar to Section 4.6.1.1.

Looking at Figure 4.16, we observe that the RLFU replacement policy provides significantly higher miss coverage than the other replacement policies when the prediction tables of the IRIP module accommodate a small number of entries. As the size of the IRIP's prediction tables increases, the miss coverage gap between RLFU and the other replacement policies shrinks because the prediction tables can store the majority of the instruction pages that produce instruction TLB misses, thus making the replacement policy of the prediction tables irrelevant.

Considering Morrigan with 3.76KB of storage budget, Figure 4.16 reveals that the LRU and Random replacement policies provide the lowest miss coverage since the former evicts useful entries based on their recency position and the latter randomly selects victims without any insight. The LFU replacement policy provides higher coverage than the LRU and Random replacement policies, highlighting that the instruction TLB miss stream correlates well with the miss frequency of the instruction pages. Finally, we observe that the RLFU policy provides the overall highest miss coverage results among all considered replacement policies. Specifically, the RLFU policy improves miss coverage over the LFU policy by 4.9%. This happens because RLFU randomly replaces one of the least recently used entries, acting like a second-chance policy for not yet frequently accessed entries.

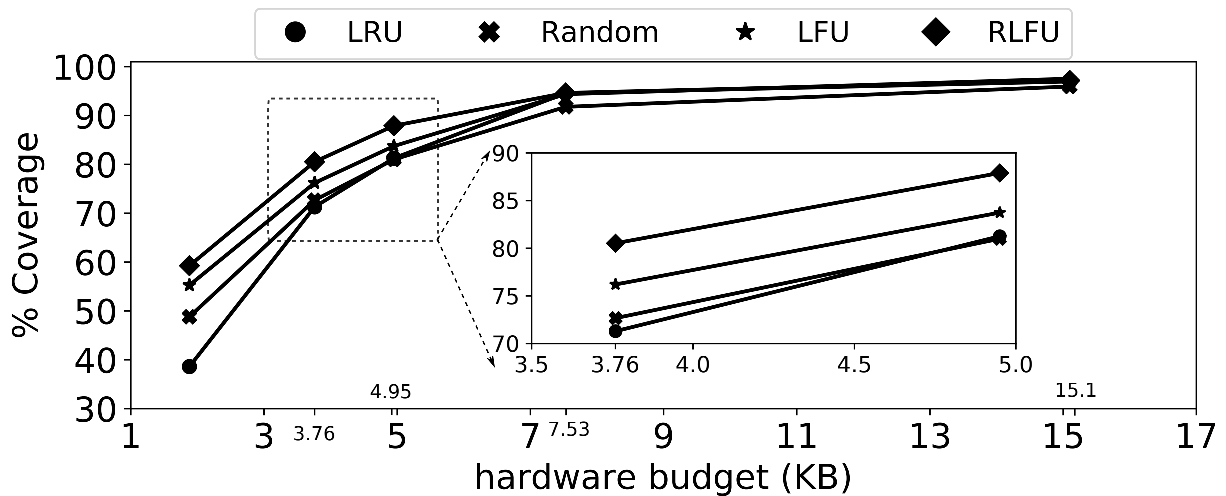


Figure 4.16: Miss coverage of Morrigan when the prediction tables of the IRIP module use different replacement policies across various storage budgets. Higher is better.

#### 4.6.1.3 Configuring IRIP

The experimental results presented in Sections 4.6.1.1 and 4.6.1.2 reveal that there is a cost-performance trade-off in the design space of Morrigan. For the rest of this section, we focus on the configuration of Morrigan with 3.76KB of storage budget, which achieves 81% miss coverage, as shown in Figure 4.15. We select this configuration because it represents an attractive point in terms of miss coverage and required storage budget.

Using the above selected version of Morrigan, we evaluate different capacities and associativities for all the prediction tables of the IRIP module. Empirically, we found the following preferred configuration: 128-entry (32 ways) PRT-S1, 128-entry (32 ways) PRT-S2, 128-entry (32 ways) PRT-S4, and a 64-entry (16 ways) PRT-S8, as shown in Table 4.2. Among the prediction tables, PRT-S8 is the smallest one because the number of instruction pages that have more than 4 and up to 8 successors is lower than the number of instruction pages that have 1, 2, and up to 4 successor pages, as shown in Section 4.3.3, and the probability of accessing a non-frequent successor page is relatively low, as illustrated in Figure 4.8. Finally, the empirically selected configuration provides a miss coverage of 76% (5% lower than the version with fully associative prediction tables).

Regarding the PB size, we consider a 64-entry PB because a PB with 16 or 32 entries provides rather poor miss coverage compared to the 64-entry PB (4%-12% reduction), whereas a 128-entry PB increases coverage by 2% compared to the 64-entry PB.

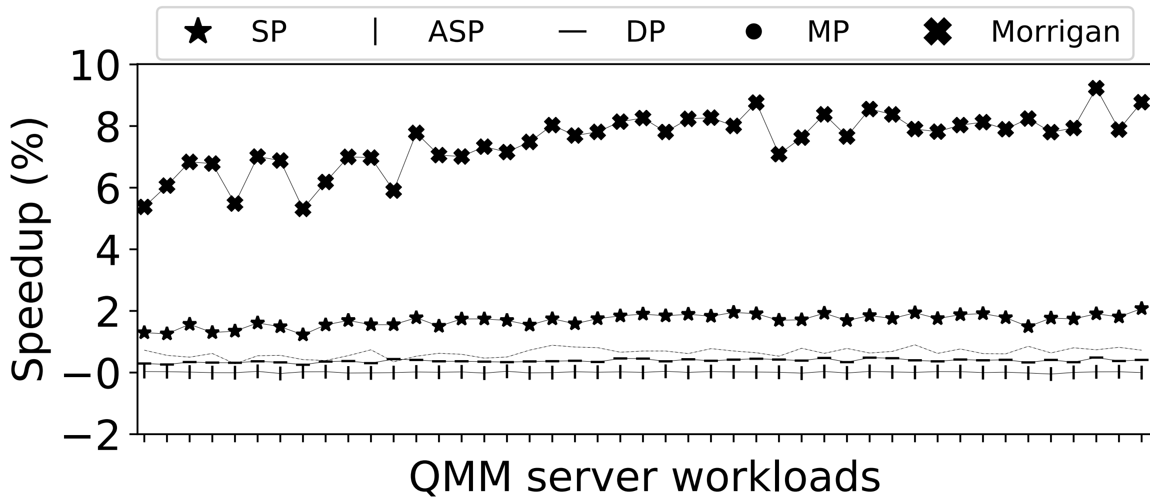


Figure 4.17: Performance comparison between Morrigan and the prior data TLB prefetchers, presented in Section 2.4.1.2. All data TLB prefetchers have been configured to match the storage budget of Morrigan. The baseline system does not apply TLB prefetching. Higher is better.

### 4.6.2 Comparison with Data TLB Prefetchers

This section compares Morrigan with previously proposed data TLB prefetchers, presented in Section 2.4.1.2, that are configured to prefetch for the instruction TLB miss stream, similar to Section 4.3.4. To make a fair comparison, we set the configuration parameters of these prefetchers in such a way that they match the storage budget of Morrigan (3.76KB).

#### Performance Comparison

Figure 4.17 shows the performance comparison between Morrigan and the data TLB prefetchers. The baseline considers the system without TLB prefetching. SP, DP, ASP, MP, and Morrigan provide a geometric speedup of 1.6%, 0.1%, 0.4%, 0.7%, and 7.6%, respectively. Morrigan significantly outperforms all previously proposed data TLB prefetchers because the QMM server workloads exhibit highly complex patterns that the data TLB prefetchers are unable to capture. Specifically, SP captures only the sequential patterns, DP and ASP experience massive conflicts in their prediction tables, and MP uses the LRU policy that fails at keeping in the prediction table the most useful instruction pages (Section 4.3.4).

In terms of PB hits provided by the two prefetching modules of Morrigan (IRIP and SDP), we measured that 93% of the prefetches that hit in the PB were triggered by the IRIP module, while the remaining 7% by SDP.

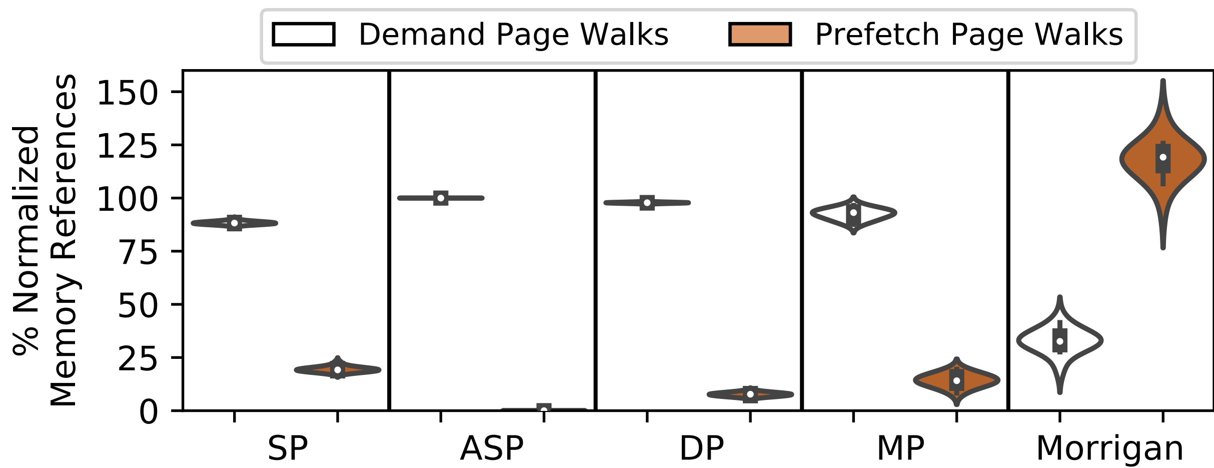


Figure 4.18: Distribution of the normalized number of memory references due to demand and prefetch page walks depicted with violin plots. The baseline does not apply TLB prefetching. Lower is better.

### Cost of Prefetching

Figure 4.18 presents the distribution of the normalized number of memory references triggered by demand and prefetch page walks for Morrigan and the prior data TLB prefetchers. The term *memory reference* refers to a page walk reference that is served by the memory hierarchy (L1, L2, LLC, DRAM), as explained in Section 2.4.1.1. Note that our methodology takes into account cache locality in page walks (Section 4.5), and a page walk memory reference is triggered only for references that miss in the MMU-Caches, which we also model. The normalization factor, 100% in Figure 4.18, is the number of memory references due to demand page walks without TLB prefetching.

Looking at Figure 4.18, we observe that SP, ASP, DP, MP, and Morrigan reduce the memory references due to demand page walks by 11%, 1%, 2%, 8%, and 69% over a baseline without instruction TLB prefetching, respectively. Regarding the prefetch page walks, SP, ASP, DP, MP, and Morrigan trigger 20%, 1%, 6%, 7%, and 117% additional memory references due to prefetch page walks with respect to the baseline, respectively.

The prior data TLB prefetchers do not reduce demand page walk memory references for instruction accesses, so they provide negligible performance improvements, as Figure 4.17 shows. They also introduce only a small number of memory references for prefetch page walks because (i) SP issues only one prefetch per instruction TLB miss, (ii) ASP and DP experience a lot of conflicting accesses in their prediction tables which does not allow

them to produce prefetch requests, and (iii) MP leverages the LRU replacement policy that fails at keeping the most useful entries in the prediction table; on prediction table lookup misses, no prefetches are issued.

While Morrigan does generate more memory references for prefetch page walks than the existing data TLB prefetchers, it achieves much higher coverage than the prior designs. Indeed, Morrigan reduces the memory references for demand page walks by 69% due to its high coverage. The vast majority of memory references due to prefetch page walks are caused by the IRIP module since the SDP module (i) issues only one prefetch at a time that requires a prefetch page walk, and (ii) is enabled only when the IRIP module is unable to issue prefetch requests, as explained in Section 4.4.1.2. However, the demand page walks are responsible for the instruction TLB performance bottleneck since they take place on the critical path of execution causing unavoidable pipeline stalls, while the prefetch page walks are performed in the background without stalling the pipeline execution.

Finally, we measure the fraction of prefetch page walk memory references served by each level of the memory hierarchy. We find that 20%, 25%, 45%, and 10% of Morrigan's prefetch page walk memory references are served by L1, L2, LLC, and DRAM, respectively. The main takeaway is that the large reduction of demand page walk memory references that Morrigan achieves, lowers the instruction address translation overhead, thus providing significant performance enhancements.

### 4.6.3 Comparing Different IRIP Designs

This section highlights the benefits of using multiple prediction tables with different numbers of prediction slots per entry for the IRIP module over the state-of-the-art approach that uses a single prediction table with a fixed number of successors per entry. To do so, we implement Morrigan-*mono* whose operation is identical to Morrigan but its IRIP module leverages a single prediction table with a fixed number of successors per entry, as the state-of-the-art MP [165] does. Since we opt to provide an ISO-storage comparison between Morrigan and Morrigan-mono, we configure the IRIP module of Morrigan-mono with a 203-entry prediction table with 8 prediction slots per entry,[1] and a 2-bit confidence counter per prediction slot to match the storage and the operation of Morrigan's IRIP module.

[1] The IRIP module of Morrigan-mono has 8 prediction slots per prediction table entry to make a fair comparison with the IRIP module of Morrigan since PRT-S8 can store up to 8 predictions per entry.

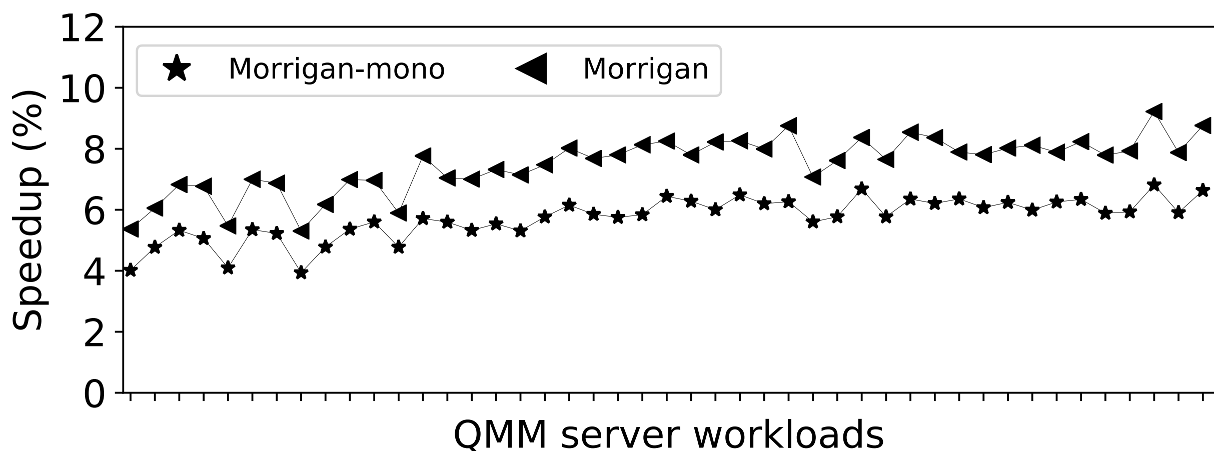


Figure 4.19: Performance of Morrigan when the IRIP module uses an ensemble of four tables (Morrigan) versus a single-table design (Morrigan-mono). The baseline system does not apply TLB prefetching. Higher is better.

Figure 4.19 reveals that Morrigan outperforms Morrigan-mono across all the QMM server workloads; the performance difference between Morrigan and Morrigan-mono is 1.9% on average. We observe this behavior because Morrigan makes better use of the available storage budget, hence tracking a much larger effective working set. Whereas Morrigan dynamically tracks the required number of prediction slots per instruction page and enables efficient transferring of entries between the prediction tables, Morrigan-mono accommodates eight prediction slots per prediction table entry. Specifically, Morrigan-mono tracks 203 entries and Morrigan effectively tracks 448 entries (128*3+64). Indeed, we find that Morrigan-mono requires 6.9KB of storage to match the performance of Morrigan having a 3.76KB storage budget.

### 4.6.4 Comparison with Other Approaches

This section compares Morrigan against other techniques that improve TLB performance and the idealized scenario that considers a Perfect TLB for instruction references (Perfect L2 TLB (inst)), as presented in Section 4.3.4. The idealized scenario is included to compare the performance of the evaluated schemes with the upper bound of the performance that can be drained out of optimizing instruction TLB accesses. Figure 4.20 presents the experimental results of this performance comparison.

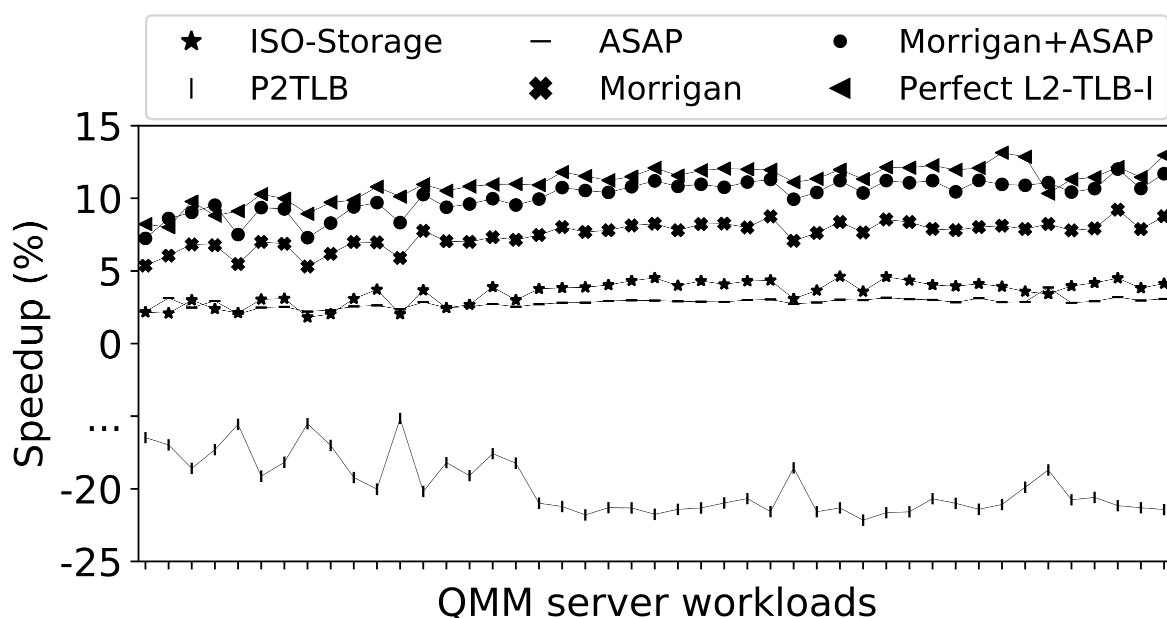


Figure 4.20: Performance comparison of Morrigan with other state-of-the-art approaches aimed at improving TLB performance. The baseline system does not apply TLB prefetching. Higher is better.

### ISO Storage

We compare our proposal, Morrigan, against a system that does not apply instruction prefetching at any TLB level but for fairness it is enhanced with an enlarged L2 TLB. Specifically, L2 TLB is augmented with 388 additional entries to match the storage budget of Morrigan (including the PB) without affecting its access time. Figure 4.20 presents that Morrigan provides 4.1% higher geometric mean speedup than the evaluated ISO-Storage scenario.

### Prefetching into the TLB

Prior TLB prefetchers [85, 165] and patents [36, 153] use a TLB buffer (named PB in this thesis) to store the prefetched PTEs. Figure 4.20 shows that placing the prefetches of Morrigan directly into the TLB (P2TLB) provides an 18.9% geometric mean performance degradation because it causes TLB pollution when the prefetches are inaccurate. Our results are consistent with prior work [48, 85, 165] stating that prefetching directly into the TLB causes pollution and performance degradation.

#### Prefetched Address Translation [188]

The work of Margaritov *et al.* [188] proposes ASAP, a $\mu$architectural scheme that lowers the latency cost of page walks by prefetching deeper levels of the radix tree page table, avoiding serialized memory references on MMU-Cache misses. Figure 4.20 shows that Morrigan outperforms ASAP by 4.8% (on average) because the MMU-Caches experience high hit rates for the QMM server workloads, thus limiting the performance gains of ASAP. We find that, on average, 1.4 memory references are required per page walk due to PSC misses. The leaf page table level always triggers a memory reference, hence only 0.4 memory references (on average) are triggered due to the other three page table levels. Therefore, the high PSC hit rate of the QMM server workloads hurts the effectiveness of ASAP.

#### Combining Morrigan with ASAP

TLB prefetching is orthogonal to techniques that aim at lowering the page walk latency. Consequently, it is natural to combine Morrigan with ASAP. The core idea is that ASAP lowers the page walk latency, thus it can be further used to accelerate the prefetch page walks of Morrigan. Figure 4.20 illustrates that combining Morrigan with ASAP improves geometric mean performance by 10.1% over a baseline without TLB prefetching, approaching the ideal performance results (Perfect L2 TLB (inst)) for most QMM server workloads. We observe such behavior because ASAP improves the timeliness of Morrigan's prefetches by accelerating the corresponding prefetch page walks.

### 4.6.5 Synergy with L1i Cache Prefetching

This section demonstrates that our proposal, Morrigan, is synergistic with state-of-the-art L1i cache prefetching. Recall that our baseline includes the next-line L1i cache prefetcher that does not cross page boundaries (Section 4.5). However, modern L1i cache prefetchers cross page boundaries, as explained in Section 4.3.5. This section studies a state-of-the-art L1i cache prefetcher, named FNL+MMA, which can cross page boundaries for instruction prefetching because it provides the highest performance among the IPC-1 L1i cache prefetchers [32] when instruction address translation is taken into account, as shown in Section 4.3.5.

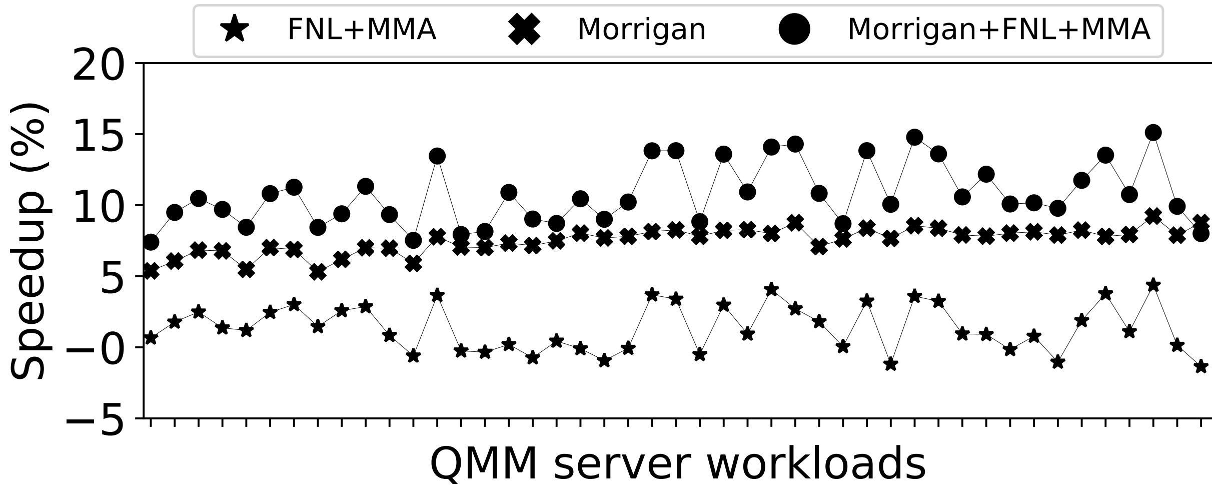


Figure 4.21: Impact of Morrigan on the performance of L1i cache prefetching. The baseline system does not apply TLB prefetching and uses the next-line L1i cache prefetcher. Higher is better.

Figure 4.21 shows the performance results of a system that uses (i) FNL+MMA, (ii) Morrigan with the next-line L1i cache prefetcher (Morrigan), as evaluated in all previous sections, and (iii) Morrigan combined with FNL+MMA L1i cache prefetcher (Morrigan+FNL+MMA). Note that the baseline corresponds to a system with a next-line L1i cache prefetcher and without prefetching at any TLB level.

Overall, FNL+MMA, Morrigan, and Morrigan+FNL+MMA provide a geometric mean speedup of 1.2%, 7.6%, and 10.9% across all QMM server workloads, respectively. We observe that the performance of the Morrigan+FNL+MMA scenario exceeds the sum of the benefits of the individual prefetchers (Morrigan, FNL+MMA). The reason why the total performance is greater than the sum of its parts is that Morrigan improves the timeliness of the FNL+MMA. Specifically, 51.7% of the beyond-page-boundary prefetches of FNL+MMA that require a page walk, hit in the PB of Morrigan+FNL+MMA, thus improving the timelines of the respective instruction prefetches. The main takeaway of this study is that Morrigan is synergistic with modern L1i cache prefetching.

### 4.6.6 Workload Colocation in SMT Cores

This section quantifies the performance of Morrigan under SMT colocation, as explained in Section 4.5. For this set of experiments, we double the size of the prediction tables of the IRIP module since Morrigan has to separately build Markov chains for two threads

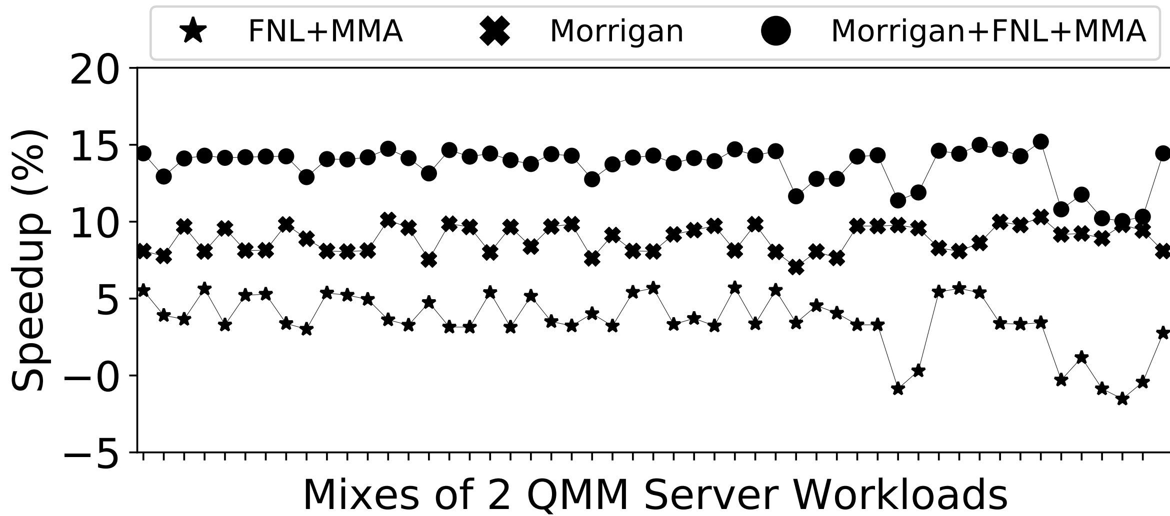


Figure 4.22: Performance of Morrigan under SMT colocation across 50 randomly chosen pairs of QMM server workloads. The baseline system does not apply TLB prefetching. Higher is better.

in the same prediction tables. As a result, the storage budget of Morrigan is increased to 7.5KBs. Our comparison includes the same set of prefetchers as in Section 4.6.5: (i) Morrigan, (ii) FNL+MMA, and (iii) Morrigan+FNL+MMA. The baseline corresponds to a system with the next-line L1i cache prefetcher and no TLB prefetching. Figure 4.22 presents the performance evaluation under SMT colocation.

The overall trends present in Figure 4.22 are consistent with the ones presented in Figure 4.21 of Section 4.6.5. However, the absolute performance gains are higher under SMT colocation, since colocating two QMM server workloads increases the pressure on the cache and the TLB hierarchy, providing a higher opportunity for instruction prefetching. Morrigan and FNL+MMA provide geometric mean speedups of 8.9% and 3.4%, respectively. Morrigan+FNL+MMA improves geometric mean performance by 13.7% because it (i) eliminates the majority of the observed instruction TLB misses, and (ii) improves the timeliness of the FNL+MMA, similar to Section 4.6.5. Finally, if the size of the prediction tables of the IRIP module is not doubled in the SMT setup, Morrigan and Morrigan+FNL+MMA improve performance by 6.4% and 11.1%, on average, respectively.

## 4.7 Related Work

### Increasing TLB Reach

Prior work increases the effective capacity of TLBs by coalescing virtually and physically contiguous PTEs into a single TLB entry [219, 221]. These approaches are (i) limited by coalescing opportunities exposed by the OS since physical contiguity is not guaranteed, and (ii) susceptible to security issues when applied for code pages because an adversary could exploit this contiguity to attack the system. In addition, Bhattacharjee *et al.* [86] propose a shared among cores last-level TLB that exploits page table locality only on demand page walks. Instead, Morrigan improves the performance of private-per-core TLBs via $\mu$architectural prefetching, exploits page table locality for both demand and prefetch walks, and does not disrupt the existing virtual memory subsystem.

### Speculative address translation

Speculation-based approaches [67, 132, 222, 296] predict the address translation of non TLB-resident pages, the processor continues executing instructions speculatively, and page walks are initiated in the background to validate whether the predicted address translations are correct. In case of valid speculation, the verification page walks overlap useful work, hiding their latency cost. Speculation-based approaches are affected by the system state since they rely on explicit virtual and physical contiguity to predict the missing address translations which is not guaranteed in systems today. Our proposal, Morrigan, exploits only virtual contiguity which comes at zero cost and is independent of the system state.

### Mitigating TLB Miss Latency

Improving the performance of the MMU-Caches [66, 82] is an effective way to reduce the latency cost of frequent TLB misses. POM-TLB [241] is a large die-stacked L3 TLB that reduces the page walk memory references to just one reference. DVMT [51] allows the application to define the appropriate page table format for an address space portion, reducing the required page walk memory references. Alternatively, hashed page tables [118, 140, 259, 300] have been proposed to resolve TLB misses faster than the conventional radix tree page tables. Morrigan is complimentary to these approaches as it eliminates instruction TLB misses via prefetching instruction PTEs ahead of demand instruction TLB accesses.

### TLB management

Typically TLBs employ a variation of the LRU replacement policy. Mirbagher-Ajorpaz *et al.* [196] propose a new predictive replacement policy for the TLB. However, TLB replacement policies aim at keeping in the TLB the most useful PTEs while TLB prefetchers proactively fetch the PTE(s) that would be requested by forthcoming memory access(es). Elnawawy *et al.* [112] identify heterogeneity in TLB behavior of data-intensive applications, *i.e.*, a few data pages have high reuse but poor temporal locality. In response, they propose Diligent TLBs, a scheme that pins in the TLB such delinquent data pages. Although effective for data pages, our analysis (Section 4.3.3) indicates that [112] needs to pin hundreds of instruction pages in the TLB to achieve significant MPKI reductions for instruction accesses; such extensive pinning raises the TLB MPKI of data pages.

### Software schemes

Compile-time optimization approaches [108, 207] modify *hugetlbfs* to place only hot functions in superpages. Moreover, OS schemes using superpages [107, 177, 304] map small code regions into superpages via superpage promotion and page table sharing. Recency-based TLB Preloading [243] builds a recency stack of PTEs in the page table to derive prefetches based on past access patterns. There are two major differences between Morrigan and [243]. First, Morrigan is a $\mu$architectural prefetcher that does not imply any page table or software modification while [243] is a software prefetching scheme that modifies the page table. Secondly, Morrigan considers access frequency for prefetching while [243] relies on recency to drive prefetching – a feature that does not correlate well with instruction TLB prefetching, as described in Section 4.3.4.

### Instruction Cache Prefetching

Numerous L1i cache prefetchers have been proposed in recent literature [32, 233]. Although effective for capturing the L1i cache miss stream, these prefetchers fall short at prefetching for the instruction TLB miss stream because they are tuned for short prefetch distances and low latencies, as the prefetched blocks are often found in the L2C or the LLC [117]. In contrast, instruction TLB misses require larger prefetch distances and incur

higher latency, caused by the serialized accesses to the memory hierarchy due to page walks. Reinman *et al.* [233] propose FDIP, a prefetching scheme that speculatively identifies instruction blocks that would potentially cause an L1i cache miss in the future and prefetch them from the lower-level caches. Intuitively, the impact of FDIP on tolerating instruction TLB misses is relatively small since it would just bring instruction PTEs into the TLB when the prefetched instruction blocks reside in memory pages different from the page where the initially missed instruction block resides. Finally, FDIP would pollute the TLB when the prefetched PTEs are inaccurate.

## 4.8 Summary

This chapter provides the first $\mu$architectural work to characterize the instruction TLB behavior of industrial server applications while providing evidence that instruction address translation is a significant performance bottleneck in servers. To mitigate this bottleneck, we propose *Morrigan*, the first ever and state-of-the-art instruction TLB prefetcher. Morrigan is a fully legacy-preserving composite prefetching module whose design is based on new reuse and locality insights of instruction TLB misses. To alleviate the instruction address translation bottleneck of server applications, Morrigan leverages two complimentary prefetch engines: (i) the Irregular Instruction TLB Prefetcher (IRIP), an ensemble of table-based hardware Markov prefetchers that dynamically build and store variable length Markov chains out of the instruction TLB miss stream while leveraging a new frequency-based replacement policy to manage their internal state, and (ii) the Small Delta Prefetcher (SDP), an enhanced sequential prefetcher that is engaged only when the IRIP module of Morrigan is unable to issue prefetch requests. Considering an extensive set of industrial server workloads, this paper demonstrates that Morrigan provides large performance enhancements by saving the majority of the instruction TLB misses while significantly reducing the references to the memory hierarchy due to demand page walks.

## 4.9 Future Work

The work presented in this chapter is the first $\mu$architectural study to characterize the instruction TLB behavior of industrial server applications and provide evidence that instruc-

tion address translation is an emerging performance bottleneck for this kind of applications. However, there is still large room for improving the performance of instruction TLB prefetching since the oracle TLB prefetcher for instruction references, presented in Section 4.3, improves geometric mean performance over Morrigan by 3.5% across all the QMM servers workloads used in this study. In addition, recent studies from leading processor vendors demonstrate that the instruction address translation will be exacerbated in the near future, essentially increasing the potential performance gains of optimizing TLB performance for instruction references.

Next, we briefly present interesting future research directions in the emerging domain of instruction address translation.

### Agile Instruction TLB Prefetching

Although effective, the design of Morrigan employs four prediction tables with a fixed number of successor pages per entry. However, this design could be improved to agilely build and store Markov chains with a variable number of successors into a single prediction table. Designing a hardware module that stores variable length Markov chains in a storage-efficient manner is a very challenging task that has the potential to spur benefits in other areas like instruction cache prefetching, branch prediction, data cache prefetching for irregular applications, and task dependency management among others.

### Branch Prediction Directed Instruction TLB Prefetching

Modern branch predictors [248] experience very high accuracy. Intuitively, the branch prediction outcome could form a useful feature for instruction TLB prefetching since it provides insights into when a branch is taken or not. This potential research direction would examine whether branch prediction can improve the accuracy of instruction TLB prefetching by filtering out small-delta (big-delta) prefetches when the branch is taken (not taken).

### ML-based Instruction TLB Prefetching

The goal of this potential research direction is to examine whether ML algorithms could be extended to accurately prefetch for the instruction TLB miss stream of big code applications.

#### Storage Medium for Prefetches and Replacement Policy

This potential research direction is similar to the one presented in Section 3.11. The fundamental idea is to examine whether the TLB PB could be reduced in size or removed by placing the prefetched instruction PTEs with high confidence directly into the TLB. In addition, this research task includes the examination of different replacement policies for the TLB and the TLB PB (if it exists).

# 5 Page Size Aware Cache Prefetching

## 5.1 Introduction

System performance continues to be limited by the Memory Wall [193, 295], *i.e.*, the discrepancy between high main memory access latencies and high processor speeds, explained in Section 2.1. To make matters worse, the increase in working set sizes of contemporary applications outpaces the growth in cache sizes, resulting in frequent main memory accesses that deteriorate system performance due to the disparity between processor and main memory speeds.

Low-latency caches can shrink the processor-memory performance gap by exploiting applications' locality to reduce the latency cost of demand memory accesses but are limited in capacity due to the overhead of implementing large SRAM structures near cores, as explained in Section 2.1.

Prefetching is a technique that hides the latency of memory accesses by proactively fetching data blocks into the cache hierarchy before a core explicitly demands them—alleviating the pressure placed on the memory subsystem by applications with large working sets that

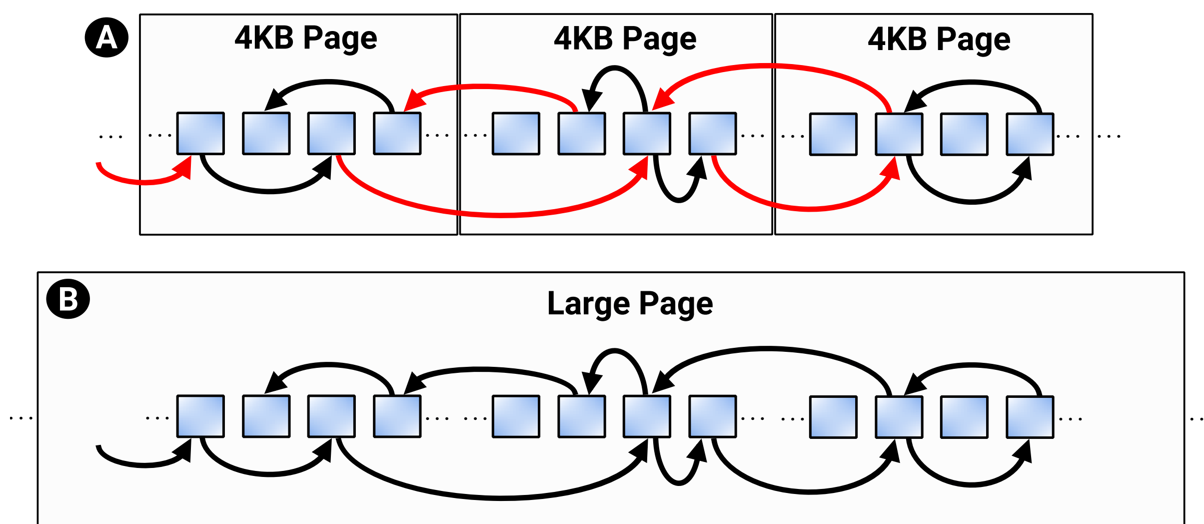


Figure 5.1: Data structure residing on multiple 4KB pages (up) and one large page (down). Arrows (black and red) illustrate the memory access patterns. Red arrows represent patterns across 4KB pages that a cache prefetcher operating on the physical address space is not allowed to prefetch for (even if it correctly identifies the patterns). Note that there are no red arrows when the data structure resides in a large page.

the caches cannot fully contain [197]. Effective prefetching has proven successful in attenuating the Memory Wall bottleneck; this is the reason why modern high-performance computing chips employ various cache prefetchers, as presented in Section 2.2.5.

Cache prefetching is a very hot research topic with myriads of prefetchers being proposed in recent literature [61, 62, 75, 76, 80, 143, 150, 172, 176, 195, 251, 262, 293, 294]. Prior cache prefetchers generally fall into two categories; spatial prefetchers and temporal prefetchers, presented in Section 2.2.4. Spatial prefetchers [61, 75, 80, 143, 172, 195, 251, 262] exploit the similarity of access patterns across different memory regions to drive prefetching decisions. In contrast, temporal prefetchers [62, 150, 176, 293, 294] do so by recording sequences of past cache misses. Although effective, temporal prefetchers have drawbacks compared to spatial prefetchers, as shown in Section 2.2.4. In summary: (i) spatial prefetchers require orders of magnitude less metadata storage compared to temporal prefetchers [61], (ii) spatial prefetchers can save compulsory misses [263] whereas temporal prefetchers are fundamentally limited to prefetch for compulsory misses, and (iii) spatial prefetchers not only save long-latency cache misses but also improve the overall system energy consumption since they increase the DRAM row buffer hit ratio [61, 143, 289].

Previously proposed spatial cache prefetchers operating in the physical address space preserve one key property: they do not permit prefetching beyond 4KB physical page boundaries as physical address contiguity is not guaranteed, *i.e.*, addresses that are contiguous in the virtual address space may be very distant in the physical address space. In addition, crossing 4KB physical page boundaries for prefetching is susceptible to security issues since an adversary could exploit it to create a side-channel [96, 128, 287]. Prefetchers are unaware of the access permissions of specific pages, thus page-crossing prefetching might allow loading data from pages the prefetcher's cache would not otherwise have access to. Indeed, a recent reverse engineering study [287] demonstrates how to exploit page-crossing prefetching at the lower-level caches to perform a side-channel attack.

Limiting lower-level spatial cache prefetchers to prefetch for intra-4KB physical page patterns limits their ability to speculate long streams of memory accesses [287]. Enabling *safe* prefetching beyond 4KB physical pages requires direct access to the TLB hierarchy and a reverse address translation [48]. These requirements pose high latency and energy overheads, hindering safe prefetching across 4KB physical page boundaries in real-world implementations.

The increase in working sets sizes of memory-intensive applications also places tremendous pressure on the TLB hierarchy [52, 55, 58, 71, 82, 117, 166, 167, 169, 175, 231, 237, 298]. This pressure results in frequent page walks that deteriorate application performance, even in the presence of dedicated architectural support for address translation [31, 33, 66, 82, 87, 91, 123, 142, 154, 171, 173, 215, 249]. The requirement for small and fast TLBs with low miss rates has led to the advent of *large pages* support in many operating systems [43, 204], architectures [7, 8, 10, 30], and virtualization enterprises [129]. For example, x86 architectures support 2MB and 1GB pages alongside standard 4KB pages to increase TLB reach.

The core idea behind the work presented in this chapter is that spatial cache prefetchers operating in the physical address space leave significant performance on the table by limiting their pattern detection within 4KB physical page boundaries when modern systems use page sizes larger than 4KB to mitigate the address translation overheads. Specifically, this chapter demonstrates that exploiting modern prevalence and support for large pages can significantly improve a system's overall performance by enabling *safe* prefetching beyond 4KB physical page boundaries when the accessed blocks reside in large pages. Figure 5.1 illustrates this opportunity by considering a generic data structure and showing its

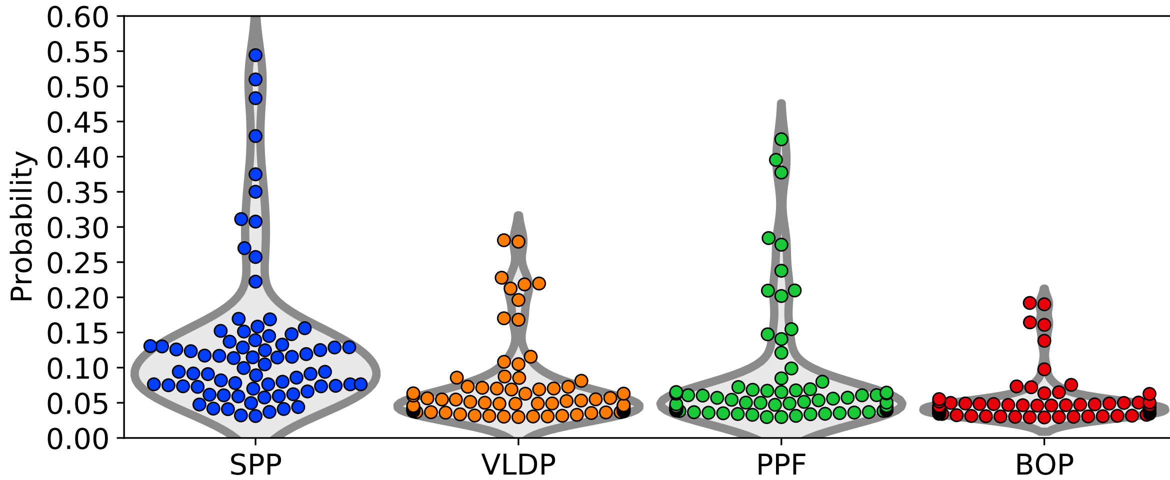


Figure 5.2: Probability distribution depicted with violin plots showing the probability for a given prefetch to be discarded because it attempts to cross 4KB physical page boundaries when the block resides in a large page, considering four state-of-the-art spatial cache prefetchers (SPP [172], VLDP [251], PPF [80], BOP [195]) and 80 memory-intensive applications, presented in Section 5.5.

memory access patterns when mapped into multiple 4KB pages **A** and a single large page **B**. Although the data structure has predictable patterns across 4KB boundaries, the cache prefetcher would not prefetch these patterns (red arrows in Figure 5.1 **A**) due to the limitation of prefetching for intra-4KB patterns. In contrast, when the data structure is mapped to a large page (Figure 5.1 **B**), the prefetcher *could* safely cross 4KB boundaries and speculate on future memory access patterns if it was aware that the block resides in a large page. However, the page size information is not inherently available to cache prefetchers operating in the physical address space.

We perform two sets of experiments to highlight the potential of leveraging the presence of large pages for improving spatial cache prefetching effectiveness. First, we demonstrate that modern systems vastly use large pages by executing a set of memory-intensive workloads spanning various contemporary benchmark suites, presented in Section 5.5.2, on a real system [44] and observing that the majority of the considered workloads heavily use large pages throughout their entire execution. Second, we quantify the missed opportunity for safely crossing 4KB physical page boundaries when the block resides in a large page (Figure 5.1 **B**) by evaluating four state-of-the-art lower-level spatial cache prefetchers, named SPP [172], VLDP [251], PPF [80], and BOP [195], measuring the num-

ber of the prefetches that these prefetchers discard due to the requirement of prefetching withing 4KB physical page boundaries when actually the accessed block resides in a 2MB page, across a set of 80 memory-intensive applications spanning various benchmark suites [19, 42, 74, 104, 117, 136]; Section 5.5.2 elaborates on the characteristics of the considered workloads. Figure 5.2 depicts the probability that a given prefetch will attempt to cross 4KB page boundaries when the block resides in a large page, but the prefetcher discards it because it is unaware that the block resides in a large page. For most workloads, 1 out of 10 prefetches are discarded due to the restriction of not crossing 4KB boundaries. At the extreme, some workloads see 1 out of 2 prefetches being discarded due to this limitation. Taking into account that cache prefetchers issue multi-million prefetches per executed workload, the probability shown in Figure 5.2 reveals that enhancing lower-level cache prefetchers with the page size information of the accessed data blocks has the potential to significantly improve cache prefetching performance due to the opportunity of safely crossing 4KB physical page boundaries when data blocks reside in large pages.

Based on our analysis findings, we propose the *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that propagates the page size information to the lower-level cache prefetchers, enabling safe prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page. PPM exploits the available address translation metadata after a first-level cache miss and directs the page size information to the lower-level cache prefetchers through the first-level caches' Miss Status Holding Registers (MSHRs). PPM operates without requiring any costly TLB lookup or reverse address translation. In addition, we highlight that PPM does not imply any modification in the design of a lower-level cache prefetcher and that is transparent to which cache prefetcher is used. For the rest of this chapter, we refer to a prefetcher that exploits PPM as *Page Size Aware Prefetcher (Pref-PSA)*.[1] It is important to note that a Pref-PSA inherently uses 4KB pages to drive prefetching decisions since PPM enables prefetching beyond 4KB physical page boundaries (when possible) without modifying the prefetcher's design.

This work further capitalizes on PPM's benefits by transparently integrating the notion of large pages into the design of any lower-level cache prefetcher.[1] We observe that doing so may positively or negatively impact performance as some workloads enjoy great benefits by making the cache prefetcher inherently use large pages while others experience performance degradation because large pages provide a coarser representation of the memory

[1] It can be any cache prefetcher operating in the physical address space.

access patterns across different data structures than standard 4KB pages. To avoid harming performance while enjoying the benefits of integrating large pages in the prefetcher's design, we design and propose a composite scheme that consists of two identical versions of the same Pref-PSA[1] that differ in only one aspect; one Pref-PSA inherently uses 4KB pages to drive prefetching while the other Pref-PSA uses large pages. Finally, the composite scheme uses adaptive selection logic based on Set-Dueling [228] to dynamically enable the most appropriate of the two competing prefetchers. For the rest of this chapter, we refer to this composite prefetcher as *Page Size Aware Prefetcher with Set Dueling (Pref-PSA-SD)*.[1]

In summary, this chapter makes the following contributions:

- This is the first study to reveal that leveraging modern prevalence and support for *large pages* can improve the effectiveness of spatial cache prefetchers operating in the physical address space due to the arising opportunity for crossing 4KB physical page boundaries when the accessed blocks reside in large pages.

- We propose *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that enables *safe* prefetching beyond 4KB physical page boundaries. Combining state-of-the-art spatial cache prefetchers SPP [172], VLDP [251], PPF [80], and BOP [195] with PPM provides single-core geometric mean speedups of 5.5%, 2.1%, 4.7%, and 2.1% over their original implementations that stop prefetching at 4KB physical page boundaries no matter the size of the page where the blocks reside, respectively, across 80 memory-intensive workloads. PPM does not imply modifications in the prefetcher's design and is transparent to which cache prefetcher is used.

- We capitalize on PPM's benefits by transparently integrating large pages into any prefetcher's implementation and designing a composite prefetcher, named *Page Size Aware Prefetcher with Set Dueling (Pref-PSA-SD)*,[1] that selects between two identical Pref-PSAs that only differ in the page size that they use to drive prefetching decisions. Our single-core evaluation shows that SPP-PSA-SD, VLDP-PSA-SD, PPF-PSA-SD, and BOP-PSA-SD outperform their original version by 8.1%, 4.0%, 5.1%, and 2.1%, respectively, across 80 memory-intensive workloads. In multi-core contexts, we report geometric mean speedups up to 7.7% across different lower-level spatial cache prefetchers and core configurations.

## 5.2 Background

All necessary background information about hardware cache prefetching and the classification of prior cache prefetchers into spatial and temporal prefetchers are presented in Sections 2.2 and 2.2.4, respectively. In addition, Section 2.4.2 presents essential background while revealing the key challenges of prefetching for the different cache levels in virtual memory systems. Modern architectural support for virtual memory systems provided by leading processor vendors is presented in Section and 2.3.3.

This section as well as the entire chapter builds on top of the concepts presented in Sections 2.2, 2.2.4, 2.4.2, and 2.3.3. Note that the information included in Section 2.4.2 is particularly important since the work presented in this chapter leverages them to improve the effectiveness of hardware cache prefetching. Therefore, understanding the concepts presented in these sections is a prerequisite for following the rest of this chapter.

This chapter focuses on x86-64 architectures and considers a system with a 2-level TLB hierarchy, a radix tree page table with 4 levels, MMU-Caches with 3 levels (called *Page Structure Caches (PSCs)* in x86-64 architectures), and a 3-level cache hierarchy, similar to Figure 2.20 in Section 2.4 and all previous chapters. In addition, it considers the most common scenario, described in detail in Section 2.2, where prefetched blocks are placed directly into the cache structure and the cache prefetcher is activated upon cache accesses.

#### 5.2.0.1 Large Pages in Practice

Even in the presence of dedicated schemes for the virtual memory subsystem, presented in Section 2.3.3, address translation is still a major performance obstacle for contemporary memory-intensive applications since they place tremendous pressure on the TLB hierarchy, resulting in frequent page walks that take hundreds of cycles to complete [188, 284]. To alleviate this bottleneck, modern systems have introduced *large pages* (also known as *superpages*), *i.e.*, pages larger than a standard 4KB page. For instance, x86-64 processors support 2MB and 1GB pages alongside standard 4KB pages. Effectively using large pages provides unique performance and energy gains. A large TLB entry accommodates the translation of a much larger contiguous memory region (*e.g.*, a 2MB page covers 512 times more memory space than a single 4KB page), increasing the effective capacity of the TLB. Finally, large pages also reduce the number of page table levels that must be traversed upon a last-level TLB miss (4 traversals with 4KB pages, 3 traversals with 2MB pages).

Modern OSes provide two mechanisms to allocate large pages. The first approach is manual since it requires the user to reserve physical memory for large pages and use the *hugetlbfs* library [46] to map specific memory segments of the application onto large pages. This approach is static and fundamentally limits the usage of large pages. In the second approach, the OS transparently allocates large pages without requiring any user involvement. Specifically, the Linux Transparent Huge Pages (THP) mechanism [43] provides automatic and transparent support for 2MB pages. However, this is not the case for pages larger than 2MB (*e.g.*, 1GB pages for x86 architectures), which still require manual allocation using the *hugetlbfs* library.[1]

## 5.3 Motivation

This section reveals that exploiting the presence of large pages in modern systems can significantly improve cache prefetching effectiveness by enabling *safe* prefetching across 4KB page boundaries. Our analysis focuses on x86 architectures with 4KB and 2MB pages since modern OSes provide automatic and transparent support for only these page sizes, as explained in Sections 5.2.0.1 and 2.3.3. In addition, we leverage the existence of large pages to improve the performance of cache prefetchers operating in the physical address space (L2C/LLC prefetchers) and not cache prefetchers driving prefetching with virtual addresses (L1d prefetchers) for the reasons explained in Section 2.4 and for two other reasons. First, L1d prefetchers issue prefetch requests using virtual addresses on every L1d access. However, the page size information is part of the address translation metadata available after the TLB access, thus waiting for the page size information upon TLB misses might harm the timeliness of L1d prefetching. Second, first-level caches necessitate simple and fast prefetchers due to their sizes and access latencies as opposed to lower-level caches that permit the implementation of sophisticated prefetchers. Finally, this chapter focuses on spatial prefetchers for the reasons outlined in Section 2.2.4, thus for the rest of this chapter we use cache prefetcher to refer to a spatial cache prefetcher placed alongside L2C or LLC, unless stated otherwise.

[1]There is a recent work [231] that aims at automatically and transparently allocating all page sizes in x86 systems (including 1GB pages).

### 5.3.1 Limitations of Existing Cache Prefetchers

Previously proposed spatial cache prefetchers operating in the physical address space preserve one key property; they assume the use of only standard 4KB pages, limiting their pattern detection to 4KB memory regions. Consequently, they do not permit prefetching beyond 4KB physical page boundaries because physical address contiguity is not guaranteed, *i.e.*, addresses that are contiguous in the virtual address space might not be contiguous in the physical address space. Moreover, prior spatial cache prefetchers stop prefetching at 4KB physical page boundaries because crossing 4KB boundaries might introduce new security vulnerabilities since an adversary could exploit page-crossing prefetching to attack the system, as explained in Section 2.4.2.

Enabling *safe* spatial prefetching beyond 4KB physical page boundaries would ideally require direct access to the TLB hierarchy to extract the virtual-to-physical mappings of the pages where the prefetched blocks reside. Doing so for spatial prefetchers operating in the physical address space requires allowing direct access from the lower-level caches to the TLB hierarchy in order to perform a reverse translation from the physical to the virtual address space. This reverse translation incurs high overheads since the reverse mappings are multi-valued functions [48].

***Finding 1.*** *There is no previously proposed μarchitectural scheme that ensures safe spatial prefetching beyond 4KB physical page boundaries for the lower-level caches.*

### 5.3.2 Opportunity for Safe Prefetching Across 4KB Boundaries

As explained in Section 5.2.0.1, systems provide support for large page sizes to reduce the severe performance and energy overheads of frequent page walks. When large pages are used, the corresponding physical mappings (physical pages) are also large, *i.e.*, the virtual pages and corresponding physical pages are of the same size. Intuitively, limiting cache prefetchers to a 4KB physical page boundary when the accessed block resides on a large page leads to suboptimal performance gains due to the missed opportunity for safely prefetching across 4KB physical page boundaries.

The first question we answer is whether modern systems practically use large pages or not. To do so, we execute a set of memory-intensive benchmarks from various contemporary benchmark suites (SPEC CPU 2006 [136], SPEC CPU 2017 [42], and GAP [74]) on an Intel

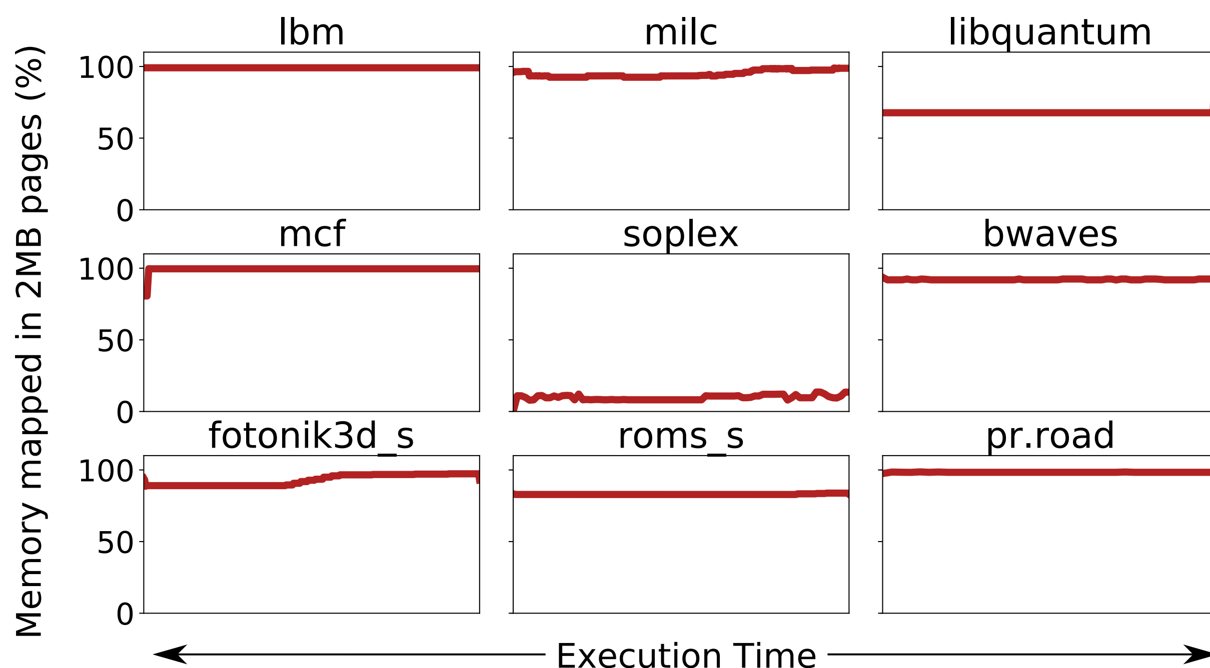


Figure 5.3: Percentage of allocated memory mapped to 2MB large pages across the entire execution of nine representative memory-intensive benchmarks from the SPEC CPU 2006 benchmark suite [136], SPEC CPU 2017 benchmark suite [42], and GAP [74] benchmark suite, running on an Intel Xeon E5-2687W machine. Most workloads preserve the high usage of 2MB large pages during their entire execution.

Xeon E5-2687W machine, collecting the usage of 4KB and 2MB pages with the page-collect tool [35] and Linux's THP mechanism [43] enabled. Figure 5.3 presents the percentage of allocated memory mapped in 2MB large pages across the entire execution of nine memory-intensive workloads. Looking at Figure 5.3, we observe that the OS gives a lot of 2MB pages to the vast majority of the evaluated workloads; the only exception is the *soplex* benchmark from the SPEC CPU 2006 benchmark suite. The main takeaway of this experiment is that most workloads heavily use 2MB pages, corroborating the conclusions of prior work [52, 231, 298]. Interestingly, we observe that most workloads preserve the high usage of 2MB large pages during their entire execution.

> ***Finding 2.*** *Modern systems heavily use 2MB pages when executing memory-intensive applications and the high usage of 2MB pages is mostly preserved throughout the entire execution of the applications.*

#### 5.3.2.1 Quantifying the Potential

Qualitatively, making lower-level cache prefetchers aware of the size of the page where the accessed blocks reside would enable safe prefetching beyond 4KB physical page boundaries when the corresponding block resides in a 2MB page, resulting in better prefetching timeliness and coverage. Removing the restriction of prefetching within 4KB physical page boundaries would allow the lower-level cache prefetchers to detect more distinct patterns, effectively increasing their coverage. Finally, the prefetchers would be able to timely prefetch patterns that cross 4KB physical page boundaries instead of waiting for an access to the next page to start issuing prefetches for the already captured patterns.

**Underlying Prefetcher**

To quantitatively answer whether or not large page exploitation can improve the effectiveness of spatial prefetching for the lower-level caches, we consider the *Signature Path Prefetcher (SPP)* [172], a confidence-based look-ahead L2C prefetcher that directs prefetched blocks into L2C or LLC depending on its internal confidence mechanism. In practice, SPP creates compressed signatures and associates them with the physical page addresses. To do so, SPP relies on two main data structures: (i) the Signature Table, a table indexed with the physical page number that stores the history of previously delta patterns per physical page as a compressed signature, and (ii) the Pattern Table, a table indexed by the signatures generated by the Signature Table that stores predicted deltas. Our motivational analysis focuses on the SPP prefetcher to demonstrate the potential benefits of enabling beyond 4KB physical page boundaries spatial prefetching by leveraging the presence of 2MB pages in modern systems since SPP provides the basis for many L2C prefetcher designs and optimizations [75, 80], and has been deployed in real-world industrial designs [126]. Our experimental campaign, presented in Section 5.6, considers three additional L2C prefetchers to highlight the versatility of the designs proposed in this chapter.

**Methodology**

To quantify whether large pages could bring benefits in spatial prefetching for the lower-level caches, we use a version of the ChampSim simulator [13, 124] that concurrently supports 4KB and 2MB pages. Section 5.5 describes in detail our simulation infrastructure.

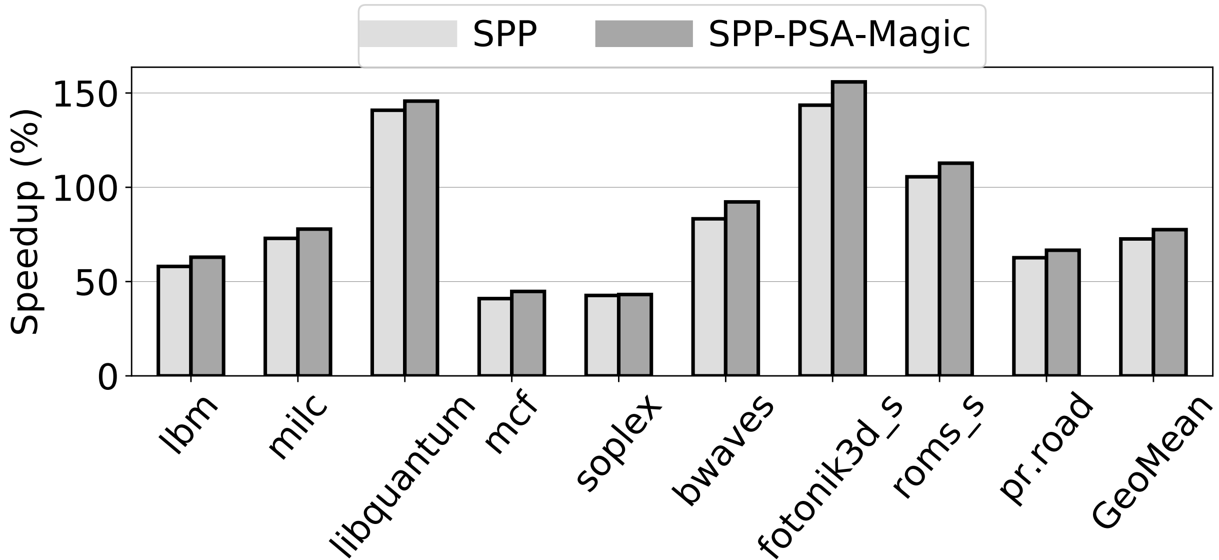


Figure 5.4: Performance comparison between the original implementation of SPP and the ideal page size aware version of SPP (SPP-PSA-Magic) across the same set of memory-intensive workloads used for the physical machine measurements, presented in Figure 5.3. The baseline system does not apply prefetching at any cache level. Higher is better.

We implement and evaluate two different versions of SPP to demonstrate the benefits of exploiting the presence of 2MB pages for enhancing lower-level cache prefetching performance. The first version corresponds to the original implementation of SPP, which stops speculation at 4KB physical page boundaries, no matter the page size of the accessed block since it does not have any notion of the page size information. The second version of SPP differs from the original in that SPP *magically* knows the page size of the accessed blocks. In practice, the idealized version of SPP stops prefetching at 4KB physical page boundaries when the block resides in a 4KB page, similar to SPP original, but it permits prefetching beyond 4KB physical page boundaries (and up to 2MB physical boundaries) when it is aware that the block resides in a 2MB page. Figure 5.4 presents the performance of the original SPP and its ideal page size aware version (SPP-PSA-Magic) over a baseline without prefetching at any cache level, similar to prior work [75, 80, 172], across the same set of memory-intensive benchmarks used for the real system measurements, presented in Section 5.3.2. Our experimental campaign, presented in Section 5.6, considers additional workloads from various benchmark suites to highlight the benefits of our proposals; Section 5.5.2 presents the complete set of workloads used in this work, describing their properties.

Looking at Figure 5.4, we observe that SPP-PSA-Magic outperforms the original version of SPP for all considered workloads. Overall, SPP-PSA-Magic improves geometric mean speedup over SPP original by 5.2%. Therefore, we conclude that magically propagating the page size information to the SPP prefetcher significantly improves its performance. The only exception is `soplex` where SPP and SPP-PSA-Magic provide similar speedups since this workload mainly operates on 4KB pages, limiting the opportunity for further performance gains by exploiting the presence of 2MB pages. In practice, SPP-PSA-Magic outperforms SPP original because it experiences better timeliness and coverage than SPP. This performance difference occurs because SPP-PSA-Magic issues prefetches that SPP would otherwise discard due to the limitation of prefetching for intra-4KB page patterns or postponing until there is an access to the prefetched block's physical page. Finally, we emphasize that SPP-PSA-Magic does not imply any modification to SPP's original design since it keeps driving prefetching decisions using 4KB indexes, similar to the original implementation of SPP.

> ***Finding 3.*** *Making cache prefetchers operating in the physical address space aware of the page size has potential for significantly improving system performance without requiring any modification in the prefetcher's design.*

### 5.3.3 Integrating Large Pages into the Design

Section 5.3.2.1 reveals that *magically* propagating the page size information to the SPP prefetcher provides large speedups across nine memory-intensive applications without implying any modification to its original design. Apart from this, what would be the performance impact of integrating 2MB pages into the design of SPP? This section answers this question, using the same baseline and the same set of workloads as Section 5.3.2.1.

The original version of SPP uses multiple data structures (Signature Table, Pattern Table) to drive prefetching, as described in Section 5.3.2.1. Only one of these structures, named Signature Table, uses the physical page number for indexing. To integrate 2MB pages into the design of SPP, we implement another version of the SPP prefetcher that differs from its original version in only one aspect; it assumes 2MB pages, not 4KB pages, to index its internal structure indexed with the physical page number, *i.e.*, the Signature Table. Consequently, the new SPP version can store deltas into the structure that stores predicted deltas (Pattern Table) that are larger than the ones stored in the corresponding structure of SPP

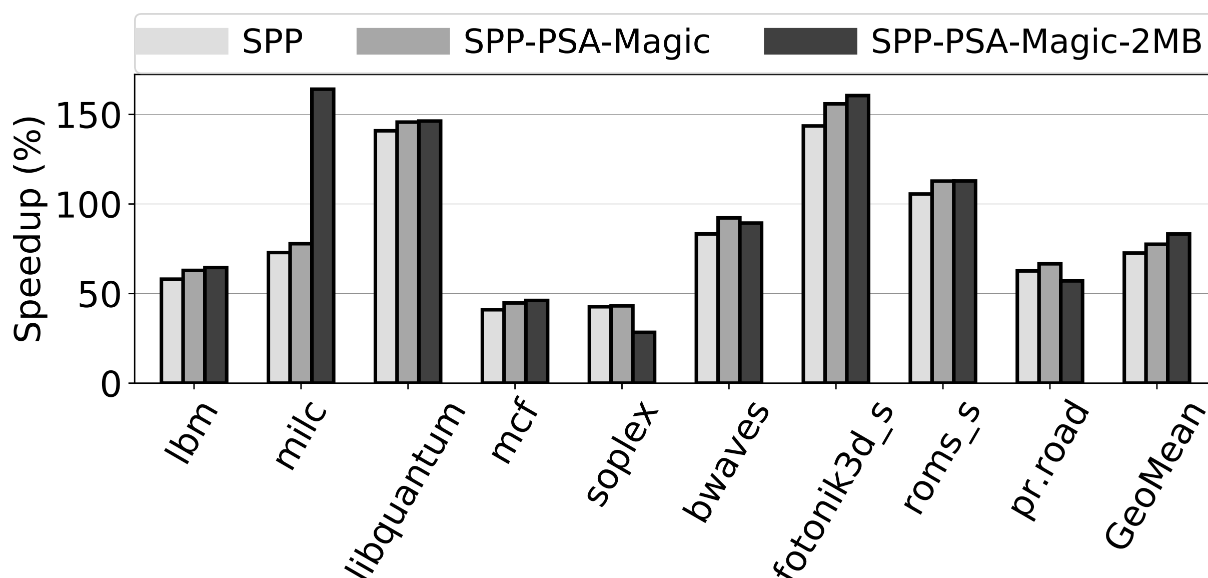


Figure 5.5: Performance comparison between the original implementation of SPP, the ideal page size aware version of SPP presented in Figure 5.4 (SPP-PSA-Magic), and deal page size aware SPP that inherently uses 2MB pages (SPP-PSA-Magic-2MB) across the same set of memory-intensive workloads used for the physical machine measurements, presented in Figure 5.3. The baseline system does not apply prefetching at any cache level. Higher is better.

original and SPP-PSA-Magic; this happens because the deltas within a 4KB page range between -64 to +64 whereas the deltas within a 2MB page range between -32768 to +32768. In addition, the compressed signature stored in the Signature Table depends on the deltas stored in the Pattern Table. Therefore, this new version of SPP has fundamental differences compared to the original implementation of SPP and SPP-PSA-Magic: (i) it reduces the aliasing in the Pattern Table due to indexing with 2MB pages at the cost of generalizing memory accesses patterns among all 4KB pages within a 2MB memory block, and (ii) it can discover patterns that SPP original and SPP-PSA-Magic fail at finding due to considering larger deltas for prefetching and/or experiencing less aliasing in the Pattern Table. We emphasize that the new SPP version that drives prefetching decisions assuming 2MB pages is magically aware of the page size to adjust its prefetching boundaries accordingly, similar to SPP-PSA-Magic of Section 5.3.2.1. For instance, if a memory block resides in a 4KB page, this new SPP version would index the Signature Table assuming that the block resides in a 2MB page but it would permit prefetching only within 4KB boundaries since it is aware that the block resides in a 4KB page. We refer to this new SPP version as SPP-PSA-Magic-2MB.

Figure 5.5 presents the performance of SPP original, SPP-PSA-Magic from Section 5.3.2.1, and SPP-PSA-Magic-2MB. We observe that SPP-PSA-Magic-2MB behaves differently across different benchmarks. For example, it provides huge speedups over both SPP and SPP-PSA-Magic for the `milc` benchmark. We observe such behavior because SPP-PSA-Magic-2MB does not suffer from aliasing in the Pattern Table and it prefetches for patterns that both SPP and SPP-PSA-Magic fail at capturing due to considering smaller deltas. For benchmarks like `libquantum`, SPP-PSA-Magic-2MB performs similarly to SPP-PSA-Magic (still greater than SPP original). However, there are benchmarks (*e.g.*, `soplex`) where SPP-PSA-Magic-2MB degrades performance over SPP original. Such behavior occurs because indexing the Signature Table with 2MB pages changes its content and the patterns it captures. Interestingly, Figure 5.5 demonstrates that indexing with 4KB pages, regardless of whether the block resides in a 4KB or a 2MB page, is sometimes better than indexing with 2MB pages since SPP-PSA-Magic outperforms SPP-PSA-Magic-2MB for some workloads (*e.g.*, `soplex`, `pr.road`). This is the case for workloads that have fine-grain address patterns (4KB-grain). In other words, when a block resides in a 2MB page it is seldom beneficial to index the prefetcher's internal data structures with 2MB pages; sometimes indexing with 4KB pages, no matter the size of the page where the accessed block resides, provides better prefetches.

> ***Finding 4.*** *Integrating 2MB pages into the design of a cache prefetcher may positively or negatively impact performance, depending on the workload. A scheme that dynamically selects between two page size aware versions of a prefetcher that drive speculation considering different page sizes has the potential to deliver outstanding benefits.*

### 5.3.4 Putting Everything Together

Sections 5.3.2.1 and 5.3.3 highlight that leveraging the presence of 2MB pages in modern systems for lower-level spatial cache prefetching has the potential to provide significant benefits. However, the reported gains assume *magically* propagating the page size information to the lower-level cache prefetchers. Realistically leveraging 2MB pages for enhancing cache prefetching performance requires (i) a scheme that propagates the page size information to the lower-level cache prefetchers, and (ii) a smart mechanism that enables the page-size aware version of the prefetcher that inherently assumes 2MB pages to drive prefetching only when it is confident that doing so would positively impact performance.

## 5.4 Design

We exploit modern prevalence and support for large pages to improve the effectiveness of cache prefetching applied in the physical address space by designing and proposing the *Page-size Propagation Module (PPM)*, the first $\mu$architectural scheme that propagates the page size information to lower-level cache prefetchers and enables *safe* prefetching beyond 4KB physical page boundaries when an accessed block resides in a large page. This section demonstrates that PPM is compatible with any lower-level cache prefetcher without implying any modification to the underlying prefetcher's implementation. For the rest of this chapter, we use *Page Size Aware Prefetcher (Pref-PSA)* to refer to a prefetcher that is aware of the page size of the accessed blocks by exploiting the PPM module. In addition, this section further capitalizes on PPM's benefits through the design of a composite $\mu$architectural scheme that transparently integrates large pages into the prefetcher's design, providing additional performance benefits at modest storage and logic costs. Sections 5.4.1 and 5.4.2 present in detail the design and the operation of the PPM module and the composite scheme that integrates large pages into the design of a cache prefetcher, respectively.

### 5.4.1 Page-size Propagation Module (PPM)

To address our analysis findings, presented in Section 5.3, we design the *Page-size Propagation Module (PPM)*, an easily implemented $\mu$architectural scheme that makes lower-level cache prefetchers aware of the page size of the accessed blocks, enabling *safe* prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page (Findings 2 and 3, Section 5.3.2). Practically, PPM augments the cache MSHRs with one additional bit indicating the page size of the corresponding accessed block. PPM does not imply any modification to the underlying prefetcher's implementation nor any costly reverse virtual to physical address translation (Finding 1, Section 5.3.1).

This section focuses on prefetching applied at the L2C to describe the design and the operation of PPM while presenting the modifications required to propagate the page size information to LLC prefetchers. We target L2C prefetching because contemporary L2C prefetchers (i) store prefetched blocks into the L2C or LLC depending on their internal confidence mechanisms, and (ii) a prefetcher placed in the L2C has a clearer view of the miss stream than an LLC prefetcher. We do not target L1d prefetchers because (i) waiting for the page

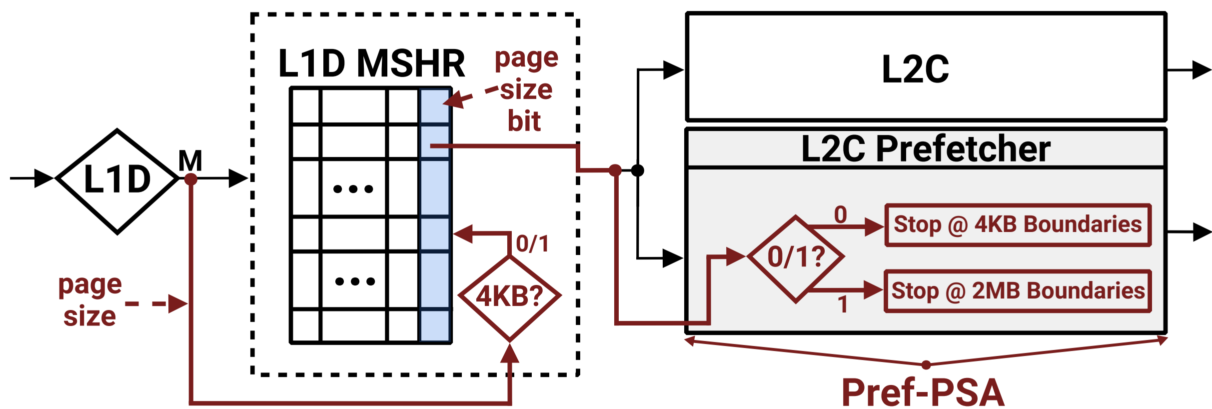


Figure 5.6: Operation of a system that employs the Page-size Propagation Module (PPM). Diamonds indicate decision points.

size information upon TLB misses might harm their timeliness since these prefetchers operate on L1d accesses, as explained in Section 5.3, and (ii) the L1d opts for low-latency accesses, hindering the implementation of sophisticated prefetchers, as explained in Sections 2.4.2 and 5.3.

#### 5.4.1.1 Implementation and Operation

The key idea behind the design of the PPM module is that first-level caches are typically implemented as virtually indexed physically tagged (VIPT) structures, as explained in Section 2.4.2, thus upon an L1d miss the size of the page where the missed block resides is available as part of the address translation metadata.

In practice, on L1d misses PPM extracts the page size information from the address translation metadata and propagates it to the L1d MSHR. To do so, we augment each L1d MSHR entry with one additional bit indicating the page size of the missed block. Since L2C prefetchers are engaged on L2C accesses, *i.e.*, L1d misses, PPM propagates the page size bit from the L1d MSHR to the L2C prefetcher via the corresponding request's stream, making the L2C prefetcher aware of the page size (Pref-PSA); we refer to such cache prefetcher as Pref-PSA.

Figure 5.6 illustrates the design and operation of a cache hierarchy enhanced with the PPM module. In practice, upon an L1d miss, PPM records the corresponding miss to the L1d MSHR coupled with one more bit annotating the page size of the corresponding block from the address translation metadata. The page size bit is either 0 or 1, indicating whether

the corresponding missed block resides in a 4KB or 2MB page, respectively. Then, Pref-PSA takes the page size bit as input and adjusts its prefetching strategy accordingly. If the page size bit is 0, Pref-PSA stops prefetching at 4KB physical page boundaries since the block resides in a 4KB page. However, if the page size bit is 1, Pref-PSA *safely* crosses 4KB boundaries since it is aware that the block resides in a 2MB page and stops prefetching at 2MB boundaries. Although Pref-PSA may prefetch across 4KB page boundaries when the accessed blocks reside in 2MB pages, it continues to index its internal prefetching structures using 4KB pages, regardless of the page size, since PPM does not imply any modification in the underlying prefetcher's design.

#### 5.4.1.2 Additional Aspects

**Storage Overhead**

PPM's implementation requires just one bit per L1d MSHR entry, assuming two concurrently supported page sizes (4KB pages and 2MB pages in x86-64 systems). Typically, L1d MSHRs have from 8 up to 64 entries, thus the storage overhead of our proposal, PPM, is considered affordable for a real-world implementation.

**Additional Page Sizes**

Although architectural support for address translation is fundamentally similar between different architectures (x86, ARM, RISC-V), some implementations support more than two page sizes. PPM is compatible with any number of concurrently supported page sizes but would require more bits stored in the L1d MSHR entries. Assuming $N$ concurrently supported page sizes, PPM needs to additionally store $\lceil \log_2 N \rceil$ bits on each L1d MSHR entry. However, the computer architect is responsible for deciding whether PPM should provide information for all supported page sizes; it might be sufficient to provide information for a subset of the supported page sizes.

**Operation on L1i Cache Misses**

Today, Linux transparently supports 2MB pages only for data, not for code. In addition, mapping code into large pages using the *hugetlbfs* library [46] might introduce security vulnerabilities [17, 18, 279, 283], as extensively explained in Section 4.5.1.1. For these

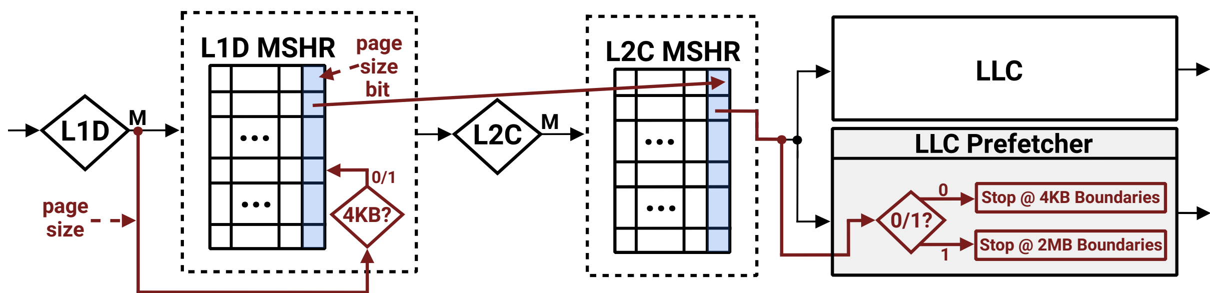

Figure 5.7: Operation of a system that leverages PPM for propagating the page size information to the LLC prefetching module. Diamonds indicate decision points.

reasons, this work considers that all instruction pages are 4KB, and do not enhance the L1i MSHR with the page size bit. We emphasize that this is not a limitation of our design, but rather an implementation choice based on the policies followed by modern systems. PPM can also be used, without any modification, to propagate the page size information to the L2C prefetching module upon L1i cache misses.

#### Applicability on LLC Prefetching

The procedure for propagating the page size information to an LLC prefetcher is fundamentally similar to the one explained in Section 5.4.1.1, but with another propagation level. First, the L2C MSHR entries should also store a bit of annotating the page size. Second, upon an L2C miss, the page size bit should be propagated from the L1d MSHR to the L2C MSHR. Finally, the L2C MSHR routes the page size bit to the LLC prefetcher. Figure 5.7 presents the operation of a system that exploits PPM for enabling safe LLC prefetching beyond 4KB physical page boundaries.

#### Security

The PPM module does not introduce new security vulnerabilities since it solely leverages the page size information which is part of the address translation metadata available after the TLB access. An adversary could not use events such as context-switches and TLB shootdowns [53, 57, 88, 181, 288] to violate the security guarantees of PPM; this would be possible if PPM was storing the page size information into a data structure and that data structure was not flushed upon TLB shootdowns and context switches.

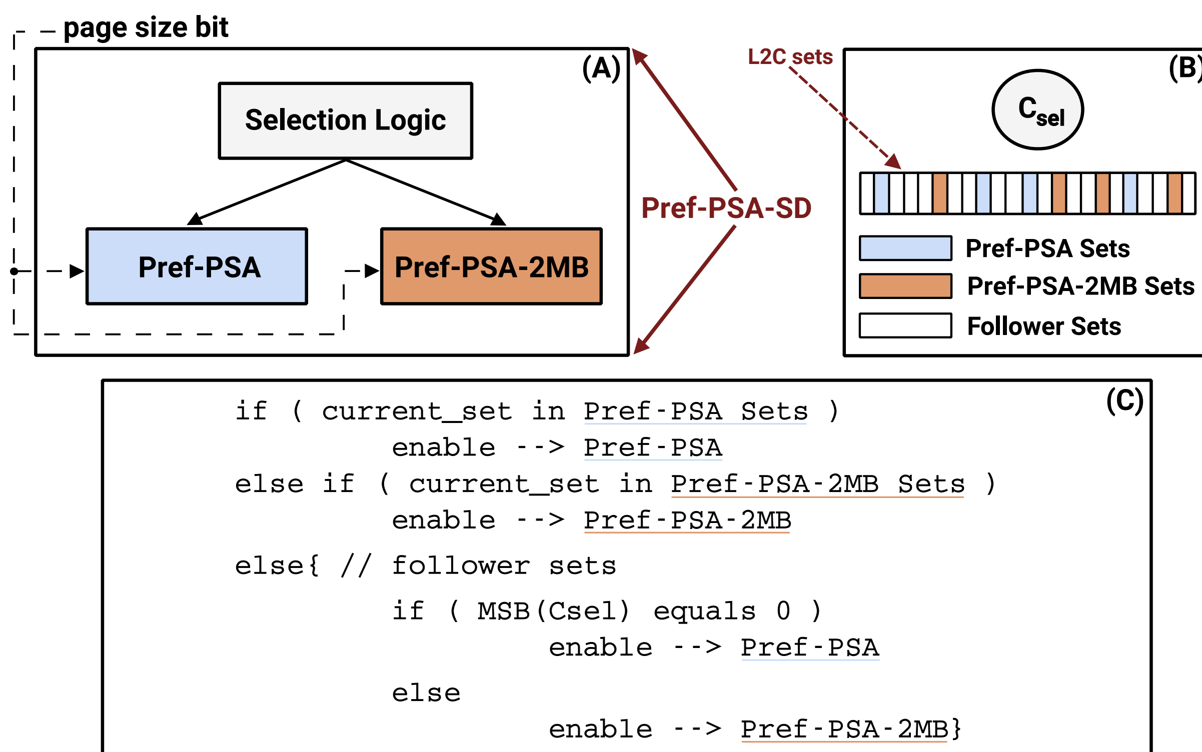


Figure 5.8: (A) L2C prefetching module comprised of two generic page size aware (PSA) prefetchers and adaptive logic that dynamically selects between them, (B) selection logic implementation, (C) operation in pseudo-code. Pref-PSA (Pref-PSA-2MB) drives prefetching assuming 4KB (2MB) pages.

## 5.4.2 Integrating Large Pages in the Design

This section builds on top of the proposed PPM module, presented in Section 5.4.1, and presumes that the page size information propagates to the L2C prefetching module. In other words, this section assumes the placement of a generic Page-size Aware Cache Prefetcher (Pref-PSA) alongside L2C.

In this section, we demonstrate how to combine an existing Pref-PSA with another page size aware version of the same prefetcher that inherently uses 2MB pages to drive prefetching decisions; we refer to this prefetcher as *Pref-PSA-2MB*. As explained in Section 5.3.3, integrating 2MB pages into the design of an L2C prefetcher may positively or negatively impact performance, depending on the workload. To address this analysis finding while avoiding performance pathologies, we implement a smart, easily implementable, and adaptive selection scheme that enables Pref-PSA-2MB only when it is highly confident that doing so will positively impact performance.

Figure 5.8 (A) depicts a high-level overview of our composite design. The L2C prefetching module consists of (i) two generic page size aware prefetchers, one inherently using 4KB pages (Pref-PSA), similar to Section 5.4.1, and another inherently using 2MB pages (Pref-PSA-2MB), and (ii) adaptive selection logic, based on Set-Dueling [228], that dynamically selects between Pref-PSA and Pref-PSA-2MB. We refer to this composite design as *Page Size Aware Prefetcher with Set Dueling (Pref-PSA-SD)*.

#### 5.4.2.1 Design of Pref-PSA-2MB

This section transparently integrates the notion of 2MB pages into the design of any L2C prefetcher. To do so, we target the internal prefetching structures of the prefetcher indexed with the physical page number (if any). The only modification necessary is that we require these data structures to be indexed assuming 2MB pages, no matter the size of the page where the accessed block resides. Although Pref-PSA-2MB assumes 2MB pages for indexing its internal structures, prefetching is permitted within the page where the trigger block resides to avoid opening side-channels (Section 2.4.2). If the prefetcher has no structure indexed with the physical page number, Pref-PSA-2MB is equivalent to Pref-PSA. Note that Pref-PSA-2MB uses predicted deltas that range between -32768 and +32768 since it assumes only 2MB pages. Therefore, Pref-PSA-2MB may, or may not, capture patterns that Pref-PSA captures, as explained in Section 5.3.3.

#### 5.4.2.2 Selection Logic

To address our last analysis finding and agilely select between Pref-PSA and Pref-PSA-2MB, we implement an adaptive selection scheme based on Set Dueling [228], a technique originally invented to select between different replacement policies within a cache structure. The selection logic that we employ for the Pref-PSA-SD scheme, presented in Figure 5.8 (B), consists of a single saturating counter, $C_{sel}$, that reflects which prefetcher to enable for the current cache access; Pref-PSA or Pref-PSA-2MB. Finally, we train both Pref-PSA and Pref-PSA-2MB on all L2C accesses since training each prefetcher only when it is selected results in insufficient training.

In practice, the selection logic clusters the L2C sets into three categories: sets dedicated to Pref-PSA, sets dedicated to Pref-PSA-2MB, and follower sets dynamically assigned to the most accurate prefetcher between Pref-PSA and Pref-PSA-2MB. We dedicate a small fraction

of the total L2C sets between the two competing prefetchers to avoid negatively impacting performance when one of the prefetchers harms performance. Empirically, we find that 32 sets are adequate for each prefetcher, similar to prior work [228].

To make our selection scheme which is based on Set Dueling properly work in the context of hardware cache prefetching, we use one bit per L2C block to annotate which prefetcher (Pref-PSA or Pref-PSA-2MB) issued the prefetch to ensure the correct updating of the saturating counter $C_{sel}$. This bit is required because the prefetched block may not be stored in the same set as the trigger block. Stress that this is not the case for cache replacement policies [228] because the cache replacement function domain is a single set. Finally, the annotation bit implies 1KB extra storage for a 512KB L2C which we consider affordable for real-world implementations.

#### 5.4.2.3 Pref-PSA-SD Operation

Pref-PSA-SD monitors the efficacy of each prefetcher by marking prefetched blocks based on the issuing prefetcher. Upon an L2C access, Pref-PSA or Pref-PSA-2MB issues prefetches for the current access based on whether or not the accessed block belongs to either prefetchers' sample set. If the corresponding block does not belong to either prefetcher's sample sets, $C_{sel}$ selects which prefetcher should be enabled. If the Most Significant Bit (MSB) of $C_{sel}$ is 0, Pref-PSA issues prefetches for the current access. Otherwise, Pref-PSA-2MB generates prefetches for the current access. The operation of Pref-PSA-SD explained above is also illustrated with pseudo-code in Figure 5.8 (C).

To update $C_{sel}$, Pref-PSA-SD takes into account the useful prefetches of the two competing prefetchers (Pref-PSA, Pref-PSA-2MB) by looking at the annotation bit, presented in Section 5.4.2.2, when there is a cache hit. In practice, a cache hit due to a prefetch issued by Pref-PSA (Pref-PSA-2MB) decrements (increments) $C_{sel}$ by one. Empirically, we found that three bits for $C_{sel}$ are adequate to identify the most useful cache prefetcher per execution phase dynamically.

Finally, no matter which prefetcher is activated, we let both Pref-PSA and Pref-PSA-2MB train on all L2C accesses and adjust their prefetching strategy accordingly. Training each prefetcher only when it is selected, as Set Dueling implies when used for cache replacement policies [228], provides poor performance gains due to insufficient training and false pattern observation, as we show in Section 5.6.1.3.

| Component | Description |
|---|---|
| **CPU Core** | 1-4 cores, 4GZ, 352-entry ROB, 4-wide |
| **L1 iTLB** | 64-entry, 4-way, 1cc, 8-entry MSHR, LRU |
| **L1 dTLB** | 64-entry, 4-way, 1cc, 8-entry MSHR, LRU |
| **L2 TLB** | 1536-entry, 12-way, 8cc, 16-entry MSHR, LRU |
| **L1 iCache** | 32KB, 8-way, 4cc, 8-entry MSHR, LRU |
| **L1 dCache** | 48KB, 12-way, 5cc, 16-entry MSHR, LRU, next line prefetcher |
| **L2 Cache** | 512KB, 8-way, 10cc, 32-entry MSHR, LRU |
| **LLC** | 2MB, 16-way, 20cc, 64-entry MSHR, LRU |
| **DRAM** | 8GB (single-core), 32GB (multi-core), 3200MT/s |
| **Branch Predictor** | hashed perceptron [273] |

Table 5.1: System simulation parameters.

## 5.5 Methodology

### 5.5.1 Performance Model

We evaluate our proposals using the ChampSim simulator [13, 124], modeling a 4-wide out-of-order processor and a three-level cache hierarchy, similar to prior work [80, 172]. Apart from the single-core evaluation, we examine the impact of our proposals in multi-core contexts. Specifically, we model 4-core and 8-core out-of-order machines. Prefetching is applied upon L2C accesses with prefetched blocks placed into the L2C or LLC, depending on the L2C prefetcher's confidence. There is no prefetching at the L1 caches, and all cache levels use the LRU replacement policy. Table 5.1 presents our experimental setup.

Prior work on spatial prefetching for the lower-level caches is limited to 4KB physical page boundaries, as explained in Section 5.3. However, modern OSes provide support for large pages, as explained in Sections 5.2.0.1 and 5.3.2. To take into account large pages, we extend the ChampSim simulator to concurrently support both 4KB and 2MB pages since

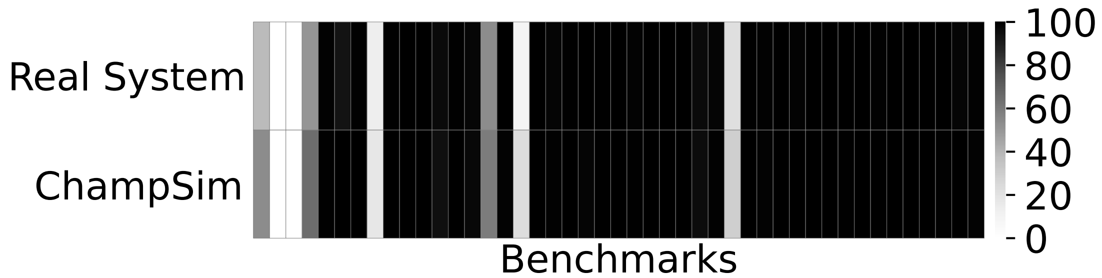


Figure 5.9: Heatmap presenting the total allocated memory mapped to 2MB pages across the considered workloads when running on Champsim and on an Intel Xeon E5-2687W machine.

the THP mechanism [43] of Linux provides automatic and transparent support only for 2MB large pages (Section 5.2.0.1). Mapping data into 1GB large pages requires manually using the *libhugetlbfs* [46] since THP does not transparently support 1GB pages. For these reasons, our evaluation considers a system that concurrently supports 4KB and 2MB pages.

We verify that our infrastructure accurately simulates multiple page sizes by measuring the usage of 4KB and 2MB pages for all benchmarks included in the SPEC CPU 2006 [136], SPEC CPU 2017 [42], and GAP [74] benchmark suites, presented in Section 5.5.2, using the page-collect tool [35] on an Intel Xeon E5-2687W machine and compare them with the corresponding usages on our simulation infrastructure. Figure 5.9 presents the comparison between the percentages of the total allocated memory mapped to 2MB pages across the considered workloads running on Champsim and on the real system. Figure 5.9 reveals that real systems heavily use 2MB pages (on average 85% of the total allocated memory is mapped to 2MB pages across the considered workloads in our system), as also shown in Section 5.3.2. Additionally, we measure that our simulation infrastructure simulates multiple page sizes within only 1.8% error compared to the real system measurements for the considered workloads.

#### 5.5.1.1 Constrained Evaluation

We test our proposals under different DRAM bandwidth configurations that roughly correspond to three commercial processors (Intel Xeon Gold [44], AMD EPYC [25], and AMD Threadripper [39]), similar to prior work [76]. Moreover, we evaluate other constrained scenarios that consider different entries in the L2C MSHR and different LLC sizes. Section 5.6.1.4 evaluates these scenarios. Finally, the multi-core evaluation uses the default configuration, presented in Table 5.1.

| Suite | Benchmarks |
|---|---|
| **SPEC CPU 2006** | gcc, bwaves, mcf, milc, cactusADM, leslie3d, gobmk, soplex, hmmer, GemsFDTD, libquantum, lbm, omnetpp, astar, wrf, sphinx3 |
| **SPEC CPU 2017** | gcc_s, bwaves_s, mcf_s, cactusBSSN_s, lbm_s, omnetpp_s, wrf_s, xalancbmk_s x264_s, cam4_s, pop2_s, leela_s fotonik3d_s, roms_s, xz_s |
| **Qualcomm** | 39 floating point and integer traces |
| **GAP** | bfs.road, cc.road, tc.urand, pr.road, bc.road, sssp.road |
| **Machine Learning** | mlpack_cf |
| **CloudSuite** | data_caching, graph_analytics, sat_solver |

Table 5.2: Complete set of workloads used for evaluation.

### 5.5.2 Workloads

We consider an extensive and diverse set of workloads that span various contemporary benchmark suites to evaluate the proposed designs of this chapter. Specifically, we use all workloads from SPEC CPU 2006 [136] and SPEC CPU 2017 [42] benchmark suites, big memory footprint workloads included in the GAP benchmark suite [74] using the `road` input graph, presented in detail in Section 3.7.4, scale-out applications from CloudSuite [117], a machine learning workload (mlpack [104]), and industrial workloads provided by Qualcomm for the 1st Contest on Value Prediction (CVP-1) [19]. For the rest of this chapter, we use QMM, SPEC, GAP, ML, and CLOUD to refer to the Qualcomm, the SPEC CPU 2006 and SPEC CPU 2017, the GAP, the mlpack, and the CloudSuite workloads, respectively.

Workloads with an LLC MPKI of at least 1 are considered memory-intensive and thus considered in our evaluation. Overall, our evaluation considers 195 different traces spanning 80 workloads. After the MPKI selection, our set of workloads includes 39 QMM workloads, 31 SPEC CPU workloads, 6 GAP, 1 ML, and 3 CLOUD workloads. All traces were

obtained using the SimPoint [218] methodology, and our evaluation reports the weighted mean speedups achieved per application. Table 5.2 presents the complete set of workloads used for evaluating the proposals of this chapter.[1]

All SPEC, GAP, CLOUD, and ML workloads run the first 250M instructions to warm up the $\mu$architectural structures and 250M instructions are executed to obtain the experimental results. For the QMM workloads, we use 50M warm-up instructions and 100M instructions for gathering results [196].

### 5.5.3 Multi-Core Evaluation

To evaluate the proposed page size exploitation techniques, presented in Section 5.4, in multi-core contexts, we randomly generate 100 mixes from our workload set. Both 4-core and 8-core evaluations use the same number of warm-up and simulation instructions as the single-core experiments. Note that we report the weighted speedup over the baseline to obviate speedup overestimation due to applications with high IPC [80, 157]. For each application running on a core, we compute the IPC in the multi-core context and the IPC in isolation on a system with the multi-core specs. Then, we compute the weighted IPC as the sum of $IPC_{\text{multi-core}}/IPC_{\text{isolation}}$ for all workloads in the mix. Finally, we normalize this sum with the weighted IPC of the baseline.

### 5.5.4 Evaluated Prefetchers

To highlight our proposals' versatility, we apply the proposed page size exploitation techniques, presented in Section 5.4, on a set of four state-of-the-art spatial L2C prefetchers: Signature Path Prefetcher (SPP) [172], Variable Length Delta Prefetcher (VLDP) [251], Perceptron-based Prefetch Filtering (PPF) [80], and Best Offset Prefetcher (BOP) [195].

We also consider the Instruction Pointer Classifier Prefetcher (IPCP) [208] in Section 5.6.1.5 to compare against state-of-the-art L1d prefetching.[2] Note that we evaluate two versions of the IPCP prefetcher; the original IPCP that prefetches for intra-4KB patterns and an enhanced IPCP version that freely crosses 4KB page boundaries.

[1] Section 5.6.1.1 presents an evaluation considering the entire SPEC CPU 2006 and SPEC CPU 2017 benchmark suites to highlight that our proposals do not harm the performance of non memory-intensive workloads.

[2] We do not compare against the Berti L1d prefetcher [205] since Berti was accepted for publication in the same conference as the paper that corresponds to this chapter.

## 5.6 Experimental Campaign

This section presents the experimental campaign of this chapter. Section 5.6.1 focuses on the single-core evaluation of our proposals. Specifically, Section 5.6.1.1 quantifies the performance benefits delivered by our proposal on both memory-intensive and non memory-intensive workloads. Section 5.6.1.2 justifies the reported performance improvements of our proposals with an in-depth analysis. In addition, we compare various implementations of the selection logic in Section 5.6.1.3, we evaluate our proposals under various constraints in Section 5.6.1.4, and we compare against state-of-the-art L1d prefetching in Section 5.6.1.5. Finally, Section 5.6.2 presents the multi-core evaluation of our proposals.

### 5.6.1 Single Core Experiments

#### 5.6.1.1 Performance

This section quantifies the single-core performance benefits of making the L2C prefetching module aware of the page size by exploiting the proposed PPM scheme, presented in Section 5.4.1, while illustrating the source of these benefits. Specifically, we quantify the performance enhancements of (i) the page size aware (PSA) versions of the considered L2C prefetchers, (ii) the page size aware versions of the L2C prefetchers that inherently use 2MB pages to index their structures (PSA-2MB), presented in Section 5.4.2, and (iii) the composite scheme (PSA-SD) that dynamically selects to enable the most appropriate between the PSA and PSA-2MB versions of the L2C prefetcher, described in Section 5.4.2.

Starting with the SPP prefetcher, Figure 5.10 reports the speedups of SPP-PSA, SPP-PSA-2MB, and SPP-PSA-SD over the original SPP implementation that is not aware of the page size information, thus it stops prefetching at 4KB physical page boundaries, across all considered workloads. Overall, SPP-PSA, SPP-PSA-2MB, and SPP-PSA-SD improve geometric mean performance over the original SPP implementation by 5.5%, 3.0%, and 8.1% across all considered workloads, respectively. The main takeaways of this evaluation are:

- SPP-PSA greatly improves performance over SPP original across the vast majority of the considered workloads (*e.g.*, `GemsFDTD`, `fotonik3d_s`, `qmm_fp_95`). We observe such behavior because SPP-PSA exploits the PPM scheme to safely cross 4KB physical page boundaries when an accessed block resides in a 2MB page, resulting in better prefetching coverage and timeliness than SPP original.

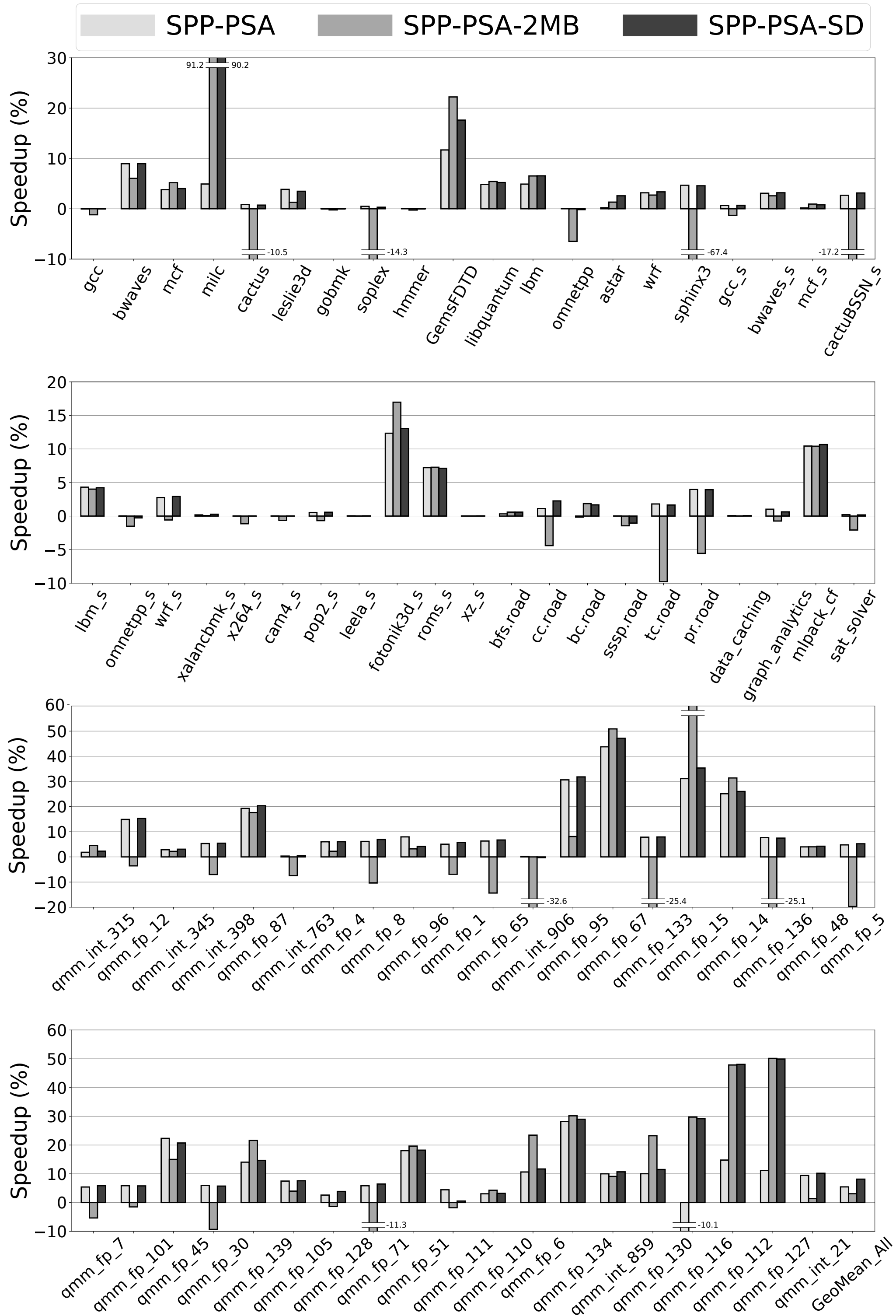


Figure 5.10: Performance comparison between the PSA, PSA-2MB, and PSA-SD versions of the SPP prefetcher. The speedups are computed over the original implementation of SPP that considers only 4KB pages and stops prefetching at 4KB physical page boundaries [172]. Higher is better.

- SPP-PSA-2MB behaves differently across different applications; for some workloads it greatly outperforms SPP original (*e.g.*, milc, qmm_fp_67) while for others it greatly degrades performance (*e.g.*, cactus, tc.road), corroborating our last analysis finding (Finding 4, Section 5.3.3) which states that it is seldom beneficial to integrate 2MB pages into the design of a lower-level cache prefetcher. We observe such behavior due to the SPP-PSA-2MB's intrinsic properties: (i) SPP-PSA-2MB indexes its internal prefetching structures assuming 2MB pages that provides a coarser representation of the access patterns than indexing with 4KB pages and less aliasing in the prediction tables, and (ii) SPP-PSA-2MB considers more strides for prefetching than SPP-Pref-PSA; SPP-PSA-2MB uses strides ranging between -32768 and +32768 while SPP-PSA uses strides ranging between -64 and +64, as explained in Section 5.3.3). Therefore, for workloads like milc, SPP-PSA-2MB outperforms SPP-PSA because it (i) does not suffer from aliasing in the prediction table, and (ii) uses strides larger than 64 that manage to detect patterns that SPP-PSA fails at finding due to only considering deltas smaller than 64. However, for benchmarks like tc.road, SPP-PSA-Magic-2MB degrades performance over SPP original and SPP-PSA because indexing its internal structure with 2MB pages erroneously generalizes different access patterns experienced by different 4KB pages residing in the same 2MB memory block into the same prefetching structure entry. In other words, indexing the prefetching structures assuming 4KB pages, regardless of whether the block resides in a 2MB page, is sometimes better than indexing assuming 2MB pages; this is the case for workloads with 4KB-grain address patterns. The main takeaway is that SPP-PSA-2MB may positively or negatively impact the performance of the system, depending on the workload, essentially motivating the design of the SPP-Pref-PSA-SD prefetching scheme.
- SPP-PSA-SD provides the overall best performance gains over SPP original. This benefit occurs because SPP-PSA-SD accurately enables the most appropriate prefetcher between SPP-PSA and SPP-PSA-2MB. For benchmarks like milc and qmm_fp_67, SPP-PSA-SD identifies that SPP-PSA-2MB is more effective than SPP-PSA and primarily enables SPP-PSA-2MB. In contrast, SPP-PSA-SD consistently enables SPP-PSA for benchmarks like sphinx3, pr_road, and qmm_fp_12 since it identifies that SPP-PSA-2MB does not provide useful prefetches for these benchmarks. Interestingly, we observe that for some workloads (*e.g.*, qmm_fp_87, cactuBSSN_s) SPP-PSA-SD yields better performance gains than its best-performing prefetcher as these workloads benefit from dynamically switching between the SPP-PSA and SPP-PSA-2MB prefetchers across different execution phases.

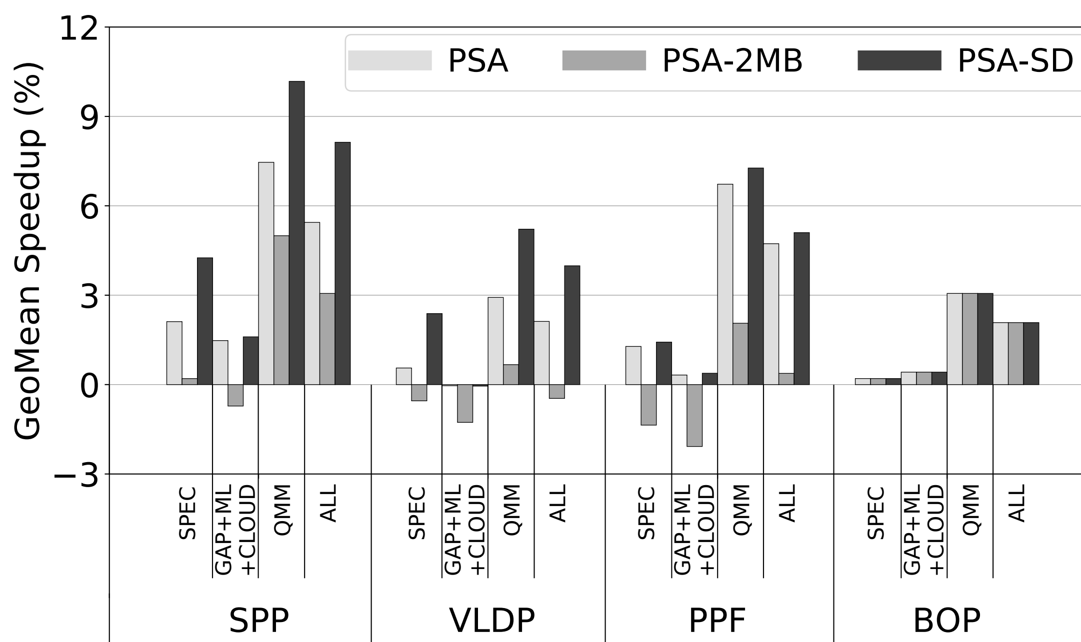


Figure 5.11: Performance comparison between the PSA, PSA-2MB, and PSA-SD versions of state-of-the-art L2C prefetchers, across all considered benchmark suites. The speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 5.10. Higher is better.

For workloads operating mainly on 4KB pages (*e.g.*, `soplex`, `hmmer`, `omnetpp`, `gcc_s`, `graph_analytics`) SPP-PSA and SPP-PSA-SD merely improve performance over SPP original because there exist only a few opportunities for safely crossing 4KB physical pages since these workloads do not use many 2MB pages. Interestingly, SPP-PSA-2MB harms performance for these workloads by erroneously generalizing access patterns to different 4KB pages within the same 2MB memory block into the same prefetching structure entry, thus using the same prefetch deltas.

### Additional Prefetchers

To demonstrate the proposed page size exploitation techniques benefit any spatial lower-level cache prefetcher, we consider the VLDP, PPF, and BOP L2C prefetchers, presented in Section 5.5.4, and evaluate their original implementation as well as their PSA, PSA-2MB, and PSA-SD versions. Figure 5.11 presents the geometric mean speedups of the PSA, PSA-2MB, and PSA-SD versions of SPP, VLDP, PPF, and BOP prefetchers across all the considered benchmark suites, presented in Section 5.5, coupled with a geometric mean across all work-

loads. The speedups are computed over the original versions of the considered prefetchers, similar to Figure 5.10. For example, the speedups of the VLDP-PSA/VLDP-PSA-2MB/VLDP-SD are computed over the original implementation of the VLDP prefetcher that considers only 4KB pages and stops prefetching at 4KB physical page boundaries.

Overall, the results reported in Figure 5.11 drive conclusions consistent with the ones reported for the SPP prefetcher in Figure 5.10: (i) the PSA version of all prefetchers greatly improves performance over the original versions across all considered benchmark suites (*e.g.*, VLDP-PSA improves geometric mean performance over VLDP original by 3.0% for the QMM workloads), (ii) the PSA-2MB version provides modest performance gains because it improves or degrades performance depending on the workload (*e.g.*, PPF-PSA-2MB improves (degrades) performance for the QMM (SPEC) workloads by 2.1% (1.3%)), and (iii) the PSA-SD version provides the best speedups since the selection logic enables the most appropriate prefetcher between PSA and PSA-2MB (*e.g.*, PPF-PSA-SD outperforms PPF original by 5.1% across all workloads). Finally, all BOP versions (PSA, PSA-2MB, PSA-SD) provide the same speedups because BOP does not use any structure indexed with the page size, causing BOP-PSA-2MB to degenerate to BOP-PSA. Therefore, BOP's PSA, PSA-2MB, and PSA-SD versions are identical, as explained in Section 5.4.2.

#### Non Memory-Intensive Workloads

Intuitively, the proposed page size exploitation techniques do not harm the performance of workloads that do not place high pressure on the cache hierarchy since they only leverage the page size information to improve the effectiveness of cache prefetching. To quantitatively address this concern, we quantify the impact of our proposals on less memory-intensive workloads by temporarily augmenting our workload set with all SPEC CPU 2006 and SPEC CPU 2017 workloads no matter their cache MPKI rates, and evaluate all considered L2C prefetchers coupled with all page size exploitation techniques. Table 5.3 presents the geometric mean performance of the PSA, PSA-2MB, and PSA-SD versions of all considered prefetchers across three different sets of workloads: (i) the non-intensive SPEC CPU 2006 and 2017 workloads (ii) the 80 memory-intensive workloads, presented in Section 5.5.2, and (iii) the 80 memory-intensive workloads plus the non-intensive SPEC CPU 2006 and SPEC CPU 2017 workloads. Looking at the performance results for Workload Set 1, we observe that our proposals (PSA, PSA-SD) do not slow down the execution of the

| Prefetcher | Workload Set 1 | Workload Set 2 | Workload Set 3 |
|---|---|---|---|
| **SPP-PSA** | 0.1% | 5.5% | 4.1% |
| **SPP-PSA-2MB** | -0.4% | 3.1% | 2.2% |
| **SPP-PSA-SD** | 0.0% | 8.1% | 6.1% |
| **VLDP-PSA** | 0.1% | 2.1% | 1.7% |
| **VLDP-PSA-2MB** | 0.4% | -0.5% | -0.3% |
| **VLDP-PSA-SD** | 0.8% | 4.0% | 3.3% |
| **PPF-PSA** | 0.1% | 4.7% | 3.4% |
| **PPF-PSA-2MB** | -0.4% | 0.4% | 0.2% |
| **PPF-PSA-SD** | 0.0% | 5.1% | 3.8% |
| **BOP-PSA** | 0.1% | 2.1% | 1.6% |

Table 5.3: Geometric mean speedups of the PSA, PSA-2MB, and PSA-SD versions of the considered prefetchers across (i) the non memory-intensive workloads from SPEC CPU 2006 and SPEC CPU 2017 benchmark suites (Workload Set 1), (ii) the 80 memory-intensive workloads of Section 5.5.2 (Workload Set 2), and (ii) the 80 memory-intensive workloads plus the non memory-intensive workloads from SPEC CPU 2006 and SPEC CPU 2017 benchmark suites (Workload Set 3). BOP-PSA-2MB and BOP-PSA-SD are excluded from this comparison because BOP does not use any structure indexed with the page size, thus all its page size aware versions are identical. The speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 5.10.

non memory-intensive SPEC workloads. In addition, it can be observed that the speedups are slightly higher when we evaluate our proposals using the 80 memory-intensive workloads (Workload Set 2) than when considering both intensive and non-intensive workloads (Workload Set 3) because the non-intensive SPEC workloads lower the reported geometric mean speedups. The main takeaway of this experiment is that page size aware lower-level cache prefetching provides significant benefits for memory-intensive workloads without negatively impacting the performance of non memory-intensive workloads.

#### 5.6.1.2 Sources of Performance Enhancements

This section justifies the benefits delivered by the proposed page size exploitation techniques. This section analyzes only the PSA and the PSA-SD versions of the prefetchers and does not consider their PSA-2MB version since the PSA-2MB version is part of the PSA-SD design that dynamically selects between the PSA and PSA-2MB versions of the prefetchers.

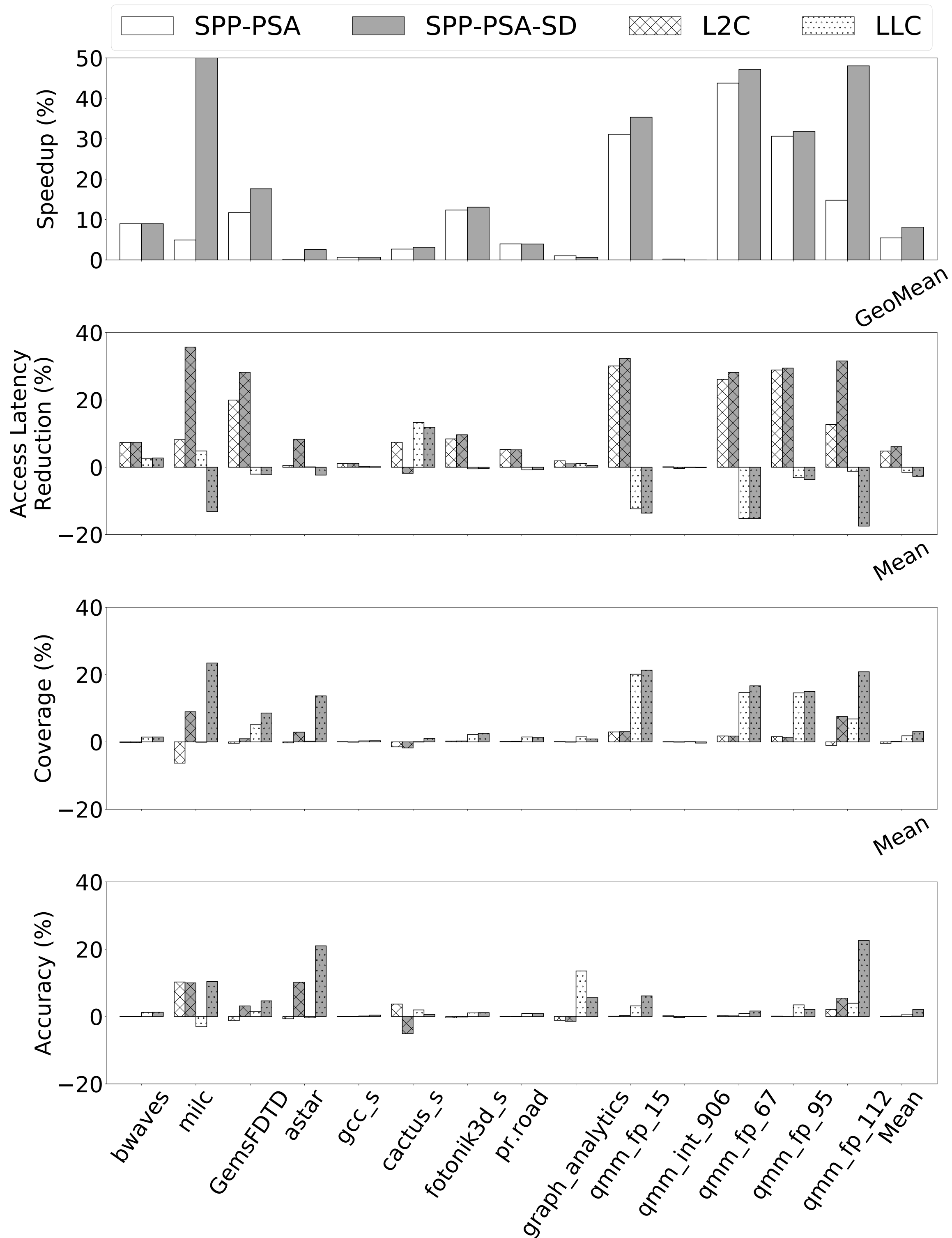


Figure 5.12: Impact of PSA and PSA-SD versions of SPP on performance, L2C/LLC access latency, L2C/LLC miss coverage, and L2C/LLC prefetching accuracy. All results are computed over the original implementation of SPP. For all considered metrics higher is better.

Figure 5.12 considers different metrics to explain the performance enhancements of our proposals on the SPP prefetcher. Specifically, we use the following metrics: (i) cache access latency (measured in cycles) to quantify the impact of our proposals on prefetching timeliness (better prefetching timeliness reduces cache access latency), (ii) miss coverage, and (iii) prefetching accuracy. Moreover, we compute these metrics for both the L2C and LLC since SPP directs prefetches in both caches, depending on its internal confidence mechanism. Finally, for this set of experiments, we consider some representative workloads across all considered benchmark suites coupled with an average across all considered workloads (Mean in Figure 5.12) for readability. Note that all metrics are computed over the original SPP version, similar to Section 5.6.1.1.

Looking at Figure 5.12 we observe that the performance gains of our proposals (PSA, PSA-SD) do not have a single root (*e.g.*, higher coverage). The speedups of SPP-PSA and SPP-PSA-SD are caused by positively impacting different metrics, depending on the workload. This is the reason why looking only at the reported averages across 80 workloads does not provide a clear understanding.

For example, SPP-PSA provides modest speedups for the `milc` benchmark, whereas SPP-PSA-SD provides massive speedups for this benchmark due to SPP-PSA-SD enabling the SPP-PSA-2MB prefetcher for this workload, as explained in Section 5.6.1.1. SPP-PSA provides modest speedups for `milc` because it improves prefetching timeliness by significantly reducing the L2C and LLC access latency costs while providing higher prefetching accuracy, at the cost of slightly reducing L2C prefetching coverage. Regarding the speedup of SPP-PSA-SD for the `milc` benchmark, we observe that it provides a large coverage increase (∼10% for L2C and ∼22% for LLC) coupled with higher prefetching accuracy (∼10% for L2C and ∼10% LLC) while reducing the L2C access latency by almost 40%. We observe a slight increase in LLC access latency because most of the LLC misses have been eliminated, and the remaining misses result in cold DRAM accesses. Similar behavior is observed for other workloads like `GemsFDTD` and `qmm_fp_112`.

For workloads like `bwaves`, `fotonik3d_s`, and `pr_road`, SPP-PSA and SPP-PSA-SD provide similar speedups because SPP-PSA-SD mainly enables SPP-PSA since SPP-PSA-2MB is not helpful for these workloads due to assuming 2MB for driving prefetching decisions. As a result, SPP-PSA and SPP-PSA-SD have almost the same impact in the metrics presented in Figure 5.12. For this group of workloads, both SPP-PSA and SPP-PSA-SD significantly reduce L2C and LLC access latencies because they experience better prefetching timeliness

Figure 5.13: Performance comparison between different implementations of the selection logic of the considered prefetchers' PSA-SD versions. Prefetcher BOP is excluded from this experiment since all BOP versions (PSA, PSA-2MB, PSA-SD) provide the same speedups because BOP does not use any structure indexed with the page size. The reported speedups are computed over the original implementation of the corresponding prefetcher, similar to Figure 5.10. Higher is better.

than SPP original while providing a slight increase in coverage and accuracy.

Both SPP-PSA and SPP-PSA-SD experience large speedups over the original implementation of SPP for workloads like `qmm_fp_15`, `qmm_fp_67`, and `qmm_fp_95`. In addition, SPP-PSA-SD outperforms SPP-PSA since it enables SPP-PSA-2MB in specific execution phases. For these workloads, both SPP-PSA and SPP-PSA-SD experience an accuracy increase of up to 10% for both L2C and LLC, a slight L2C coverage increase (<10%), a massive LLC coverage increase (>13%), a massive reduction in L2C access latency due to better prefetching timeliness, and a small increase in LLC latency, because most LLC misses have been eliminated thus the remaining misses result in cold DRAM accesses.

For workloads like `gcc_s`, `graph_analytics`, and `qmm_int_906` both SPP-PSA and SPP-PSA-SD merely improve performance over SPP original because these workloads mainly operate on 4KB pages, thus there is no potential for high performance gains. Consequently, they have a small impact on the metrics of Figure 5.12, and the proposed page size exploitation techniques merely improve their performance, as shown in Figure 5.10.

Finally, this section focuses on the SPP prefetcher to demonstrate the sources of performance enhancements delivered by the proposed page size exploitation schemes (Figure 5.12). We observe similar behavior for the rest of the evaluated cache prefetchers (VLDP, PPF, BOP).

#### 5.6.1.3 Different Selection Logic Implementations

This section highlights the benefits of the proposed selection logic, presented in Section 5.4.2, by comparing it against alternative selection logic implementations. Specifically, Figure 5.13 presents the geometric mean speedups of the PSA-SD versions of all considered L2C prefetchers, across three different selection logic implementations: (i) original implementation of Set-Dueling [228] that trains the PSA and PSA-2MB only when they are selected (SD-Standard), (ii) page size based selection scheme (SD-Page-Size) where the selection logic blindly enables the PSA (PSA-2MB) version of the prefetcher when the accessed block resides in a 4KB page (2MB page), and (iii) the proposed selection logic implementation (SD-Proposed). In addition, we evaluate an ISO storage scenario that doubles the storage budget of the original prefetchers' implementations to match the budget of the PSA-SD versions and the cost of the annotation bit (Section 5.4.2.2). Prefetcher BOP is excluded from this experiment since BOP-PSA and BOP-PSA-SD are the same, as shown in Section 5.6.1.1. Finally, the speedups are computed over the original versions of the considered L2C prefetchers, similar to Figure 5.11.

Figure 5.13 reveals that SD-Proposed provides the overall highest speedups across all prefetchers (*e.g.*, SD-Proposed outperforms the other selection logic implementations by up to 6.4% for SPP). In addition, we observe that SD-Standard provides lower speedups than SD-Proposed; this happens because SD-Standard trains the PSA and PSA-2MB versions of the prefetchers only when they are selected, whereas SD-Proposed trains both prefetchers across all accesses. Moreover, we find that SD-Page-Size provides good speedups but still performs worse than the proposed selection logic implementation (SD-Proposed). We observe such behavior because indexing the internal prefetching structures of the considered prefetchers with 2MB pages sometimes loses important information due to a coarser representation of patterns, leading to suboptimal prefetching decisions. In other words, blindly considering the page size to enable one of the PSA and PSA-2MB versions is seldom beneficial. Sometimes it is better to assume 4KB (2MB) pages for indexing the internal prefetching structures even if the accessed block resides in a 2MB (4KB) page.[1] This happens when 2MB pages accommodate data structures with orthogonal memory access patterns; in these cases, the prefetcher is more effective at capturing the memory access patterns of the differ-

[1]No matter which page size is considered for indexing, prefetching is permitted within the page where the accessed block resides.

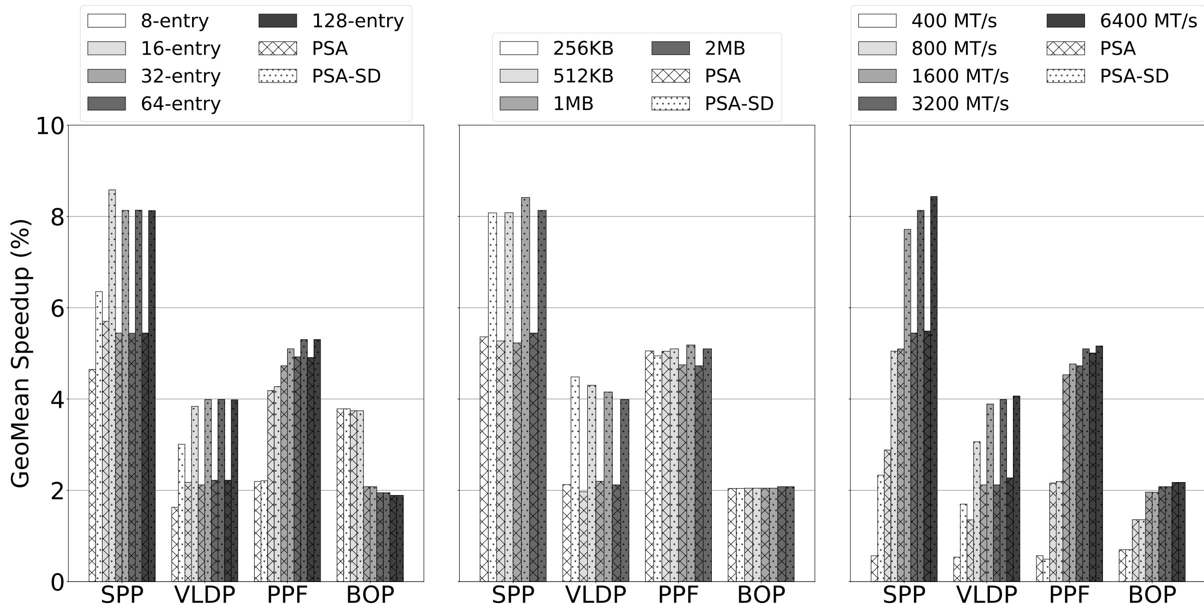


Figure 5.14: Impact of L2C MSHR size (A), LLC size (B), and DRAM bandwidth (C) on the performance of the PSA and PSA-SD versions of the considered L2C prefetchers. Results are computed over the original implementations of the considered L2C prefetchers. Higher is better.

ent structures by internally considering 4KB pages since fewer data structures are clustered within one 4KB page than a 2MB page. Finally, the ISO storage scenario's speedups reveal that doubling the prefetchers' size merely improves performance, highlighting the benefits of our proposals.

#### 5.6.1.4 Constrained Evaluation

This section quantifies the impact of various constraints on the performance of the PSA and PSA-SD versions of all considered prefetchers. Figure 5.14 presents the impact on performance for various L2C MSHR sizes, LLC sizes, and DRAM bandwidths roughly corresponding to various commercial processors, as presented in Section 5.5.1.1. The speedups are computed over the prefetchers' original versions, similar to Section 5.6.1.1.

Results presented in Figure 5.14 (A) and (B) reveal that no matter the L2C MSHR size and the LLC capacity the PSA and the PSA-SD versions of all considered prefetchers consistently provide large speedups over the original versions of the prefetchers. For instance, even when the L2C MSHR has 8 entries, SPP-PSA and SPP-PSA-SD improve geometric mean speedup over SPP original by 4.6% and 6.4%, respectively.

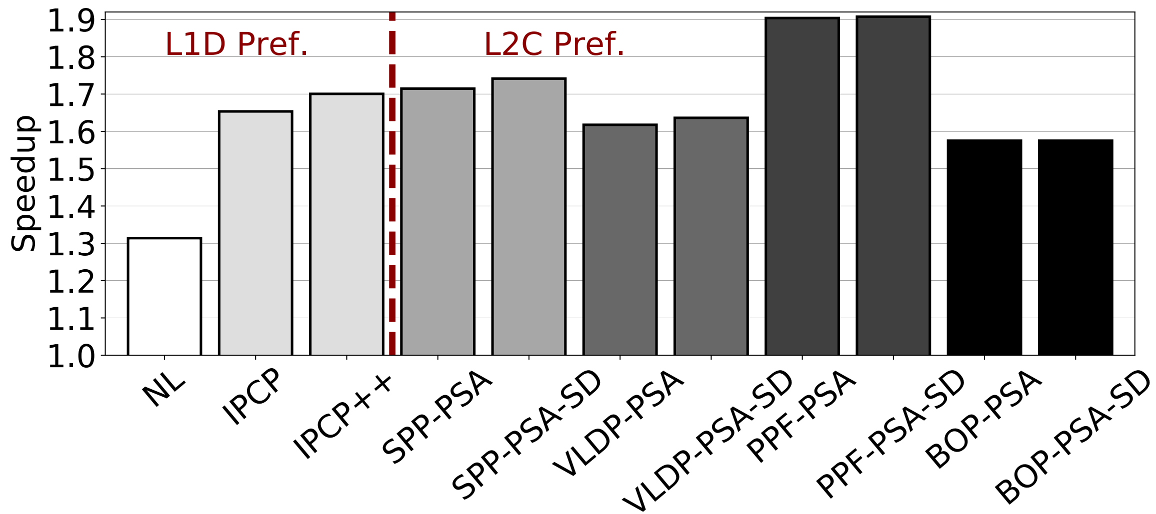


Figure 5.15: Comparison with state-of-the-art L1d prefetching. NL refers to a next-line L1d prefetcher. IPCP is the state-of-the-art L1d prefetcher. IPCP++ refers to a version of IPCP that freely crosses 4KB page boundaries for prefetching. Higher is better.

Regarding the impact of DRAM bandwidth, presented in Figure 5.14 (C), on the speedups of our proposals, we observe that the PSA and PSA-SD versions of the prefetchers consistently improve performance over their original versions, even when DRAM bandwidth is 400 MT/s. The main takeaway of this evaluation is that exploiting the page size information to safely prefetch across 4KB physical page boundaries provides large gains even in bandwidth-constrained scenarios.

#### 5.6.1.5 Comparison with L1d Prefetching

This section compares the PSA and PSA-SD versions of all considered L2C prefetchers with the Instruction Pointer Classifier Prefetcher (IPCP) [208] which is a state-of-the-art L1d prefetcher. We evaluate two versions of IPCP: the first (IPCP in Figure 5.15) stops prefetching at 4KB page boundaries, and the second (IPCP++ in Figure 5.15) is allowed to cross 4KB page boundaries for prefetching only when the page where the prefetched block resides is TLB resident, as explained in Section 2.4.2. Both IPCP and IPCP++ apply prefetching using virtual addresses since they are placed alongside L1d. We also evaluate a simple next-line prefetcher for reference. Figure 5.15 presents the speedups of the considered prefetchers across all 80 memory-intensive workloads of Section 5.5.2. The baseline system does not use prefetching at any cache level.

Looking at Figure 5.15, we observe that the version of IPCP that crosses 4KB page boundaries for prefetching (IPCP++) delivers higher speedup than IPCP that stops prefetching at 4KB page boundaries. Overall, IPCP++ improves geometric mean speedup over IPCP by 4.6%. We observe such behavior because IPCP++ experiences higher coverage and better timeliness than IPCP due to 4KB-crossing prefetching. However, the PSA and PSA-SD versions of SPP and PPF outperform both IPCP and IPCP++. For example, SPP-PSA-SD and PPF-PSA-SD provide 9.0% (4.4%) and 24.6% (20.0%) higher speedups than IPCP (IPCP++), respectively. In addition, the versions of VLDP and BOP that exploit the page size information to safely cross 4KB physical page boundaries for prefetching provide speedups slightly lower than IPCP and IPCP++. The main takeaway of this experiment is that page size aware L2C prefetching delivers equal or higher performance enhancement than state-of-the-art L1d prefetching with and without crossing page boundaries.

### 5.6.2 Multi-Core Experiments

This section presents the performance benefits delivered by our proposals (PSA, PSA-SD) to all considered L2C prefetchers in multi-core contexts. Figures 5.16 and 5.17 illustrate the distribution of the speedups of the PSA and PSA-SD versions, across the SPP, VLDP, PPF, and BOP prefetchers, in a 4-core and an 8-core context, respectively. Both 4-core and 8-core experiments use 100 random mixes, as explained in Section 5.5.2. Finally, the speedups reported in Figures 5.16 and 5.17 are computed over the original versions of the considered L2C prefetchers, similar to Section 5.6.1.1.

Our multi-core evaluation reveals that both PSA and PSA-SD versions of all considered L2C prefetchers provide large performance benefits for the vast majority of the 4-core and 8-core mixes. For example, SPP-PSA and SPP-PSA-SD provide a geometric mean speedup of 5.6% and 7.7% over the original SPP implementation across 100 randomly generated 4-core mixes. However, in the 8-core context, we observe that the PSA and PSA-SD versions of all the prefetchers deliver lower performance enhancements than in the 4-core context. This happens because both 4-core and 8-core evaluations use the same DRAM configuration (Table 5.1 in Section 5.5). Therefore, there is less opportunity for performance improvements by exploiting the page size information for prefetching in the 8-core context due to limited bandwidth compared to the 4-core context.

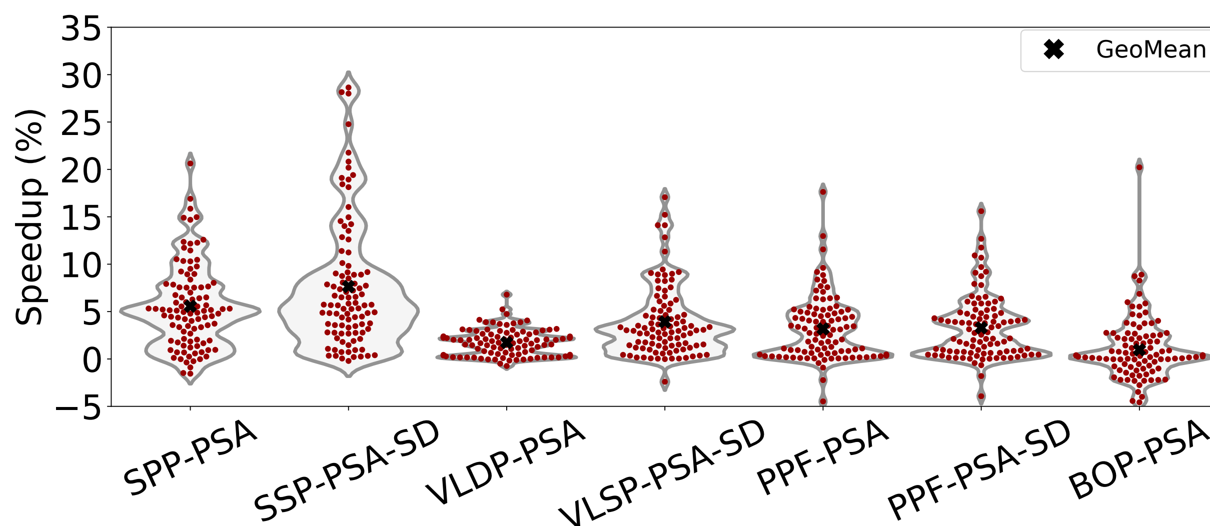


Figure 5.16: Distribution depicted with violin plots showing the 4-core speedups of the PSA and PSA-SD versions of the considered L2C prefetchers (SPP, VLDP, PPF, BOP) across 100 4-core mixes. Higher is better.

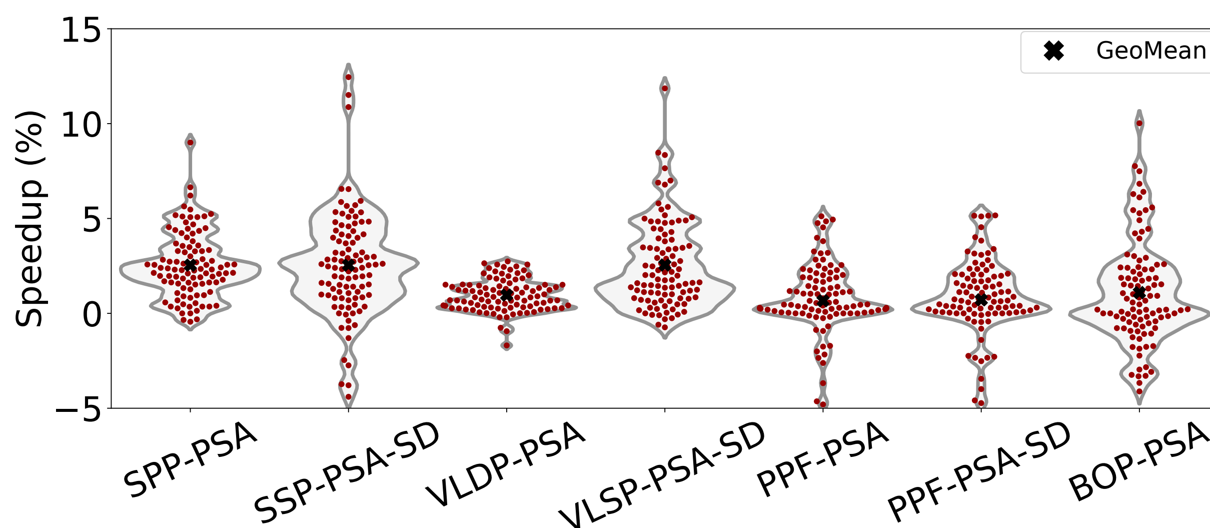


Figure 5.17: Distribution depicted with violin plots showing the 8-core speedups of the PSA and PSA-SD versions of the considered L2C prefetchers (SPP, VLDP, PPF, BOP) across 100 8-core mixes. Higher is better.

## 5.7 Summary

This chapter provides the first ever $\mu$architectural study to characterize the potential of enabling safe prefetching beyond 4KB physical page boundaries. In addition, it is the first work to reveal that leveraging modern prevalence and support for large pages can improve the effectiveness of spatial cache prefetchers operating in the physical address space due to the arising opportunity for crossing 4KB physical page boundaries when the accessed blocks reside in large pages. In response, we design and propose the *Page-size Propagation Module (PPM)*, a $\mu$architectural scheme that exploits the prevalence of large pages in modern systems to enable *safe* prefetching beyond 4KB physical page boundaries when the accessed blocks reside in large pages. PPM is a fully-legacy preserving $\mu$architectural scheme that incurs almost zero storage and logic costs and does not introduce new security vulnerabilities. We capitalize on PPM's benefits by designing and proposing a module comprised of two page size aware prefetchers that inherently assume different page sizes to drive prefetching while using adaptive selection logic to enable the most appropriate page size aware prefetcher per cache access. This composite module is transparent to which cache prefetcher is used. Across an extensive set of academic and industrial workloads, we demonstrate that the proposed page size exploitation techniques provide significant performance enhancements on various state-of-the-art lower-level cache prefetchers while not harming the performance of non memory-intensive workloads. Finally, we firmly believe that the findings and the proposals of this chapter have the potential to impact next-generation industrial designs and initiate additional research in the domain of page size aware cache prefetching.

## 5.8 Future Work

The work presented in this chapter is the first $\mu$architectural study to provide evidence that propagating the page size information to the lower-level cache prefetchers has the potential to provide outstanding performance enhancements when large pages are used. However, hardware cache prefetching is an endless opportunity domain, as explained in Section 1.2, thus this work is just the first step towards fully exploiting the page size information (and potentially other address translation metadata) for enhancing the performance of cache prefetching.

Next, we briefly present interesting future research directions in the domain of cache prefetching that might trigger prefetches beyond 4KB page boundaries.

### Page-size Aware Cache Prefetching & 1GB Pages

Superpages are pages that are larger than standard 4KB pages. For example, x86-64 architectures provide support for both 2MB and 1GB superpages alongside base 4KB pages. The work presented in this chapter focuses on 2MB superpages since Linux provides automatic and transparent support only for 2MB superpages through the THP mechanism [43]; 1GB superpages still require manually using the *libhugetlbfs* [46]. However, the increase in working set sizes of applications outpaces the growth in cache sizes, further aggravating the Memory Wall bottleneck. In response, the research community is actively working on providing automatic and transparent support for 1GB superpages. Specifically, there is a recent work [231] that aims at automatically and transparently allocating all page sizes in x86 systems (including 1GB pages).

When 1GB pages are used, there is an opportunity for aggressive prefetching at the lower-level caches by leveraging the proposed PPM module since 1GB pages are 512 times bigger than 2MB pages. This potential research direction would examine the benefits of exploiting the PPM module for applications that practically use 1GB pages.

### TLB Prefetching & L1d Prefetching Beyond 4KB Boundaries

Prefetchers operating alongside the L1d cache have direct access to the TLB hierarchy, thus they could easily cross 4KB page boundaries and search for the translation on the TLB hierarchy. In practice, doing so is more complicated, as explained in Section 2.4.2. What should L1d prefetchers do when the translation of the page where the prefetched block resides is not TLB resident? Even in the case of oracle L1d prefetching, triggering a long-latency page walk for fetching the translation of the page where the prefetched block resides would greatly harm the timeliness of L1d prefetching since L1C prefetchers require quick turnaround times on memory accesses due to the sheer amount of requests seen at the first-level caches. For these reasons, state-of-the-art L1C prefetchers [208] typically permit prefetching within standard 4KB page boundaries. This promising research direction would design a synergistic TLB prefetcher aimed at improving the timeliness of L1d page-crossing prefetching.

# Conclusions

Virtual memory amplifies the discrepancy between processor and main memory speeds while increasing the energy consumption of the system due to the requirement of translating virtual addresses into physical ones upon memory accesses, even in the presence of sophisticated architectural support for address translation. The advent of contemporary applications with massive data and code footprints further exacerbates the Memory Wall bottleneck since they place tremendous pressure on the TLB and cache hierarchies.

This dissertation provides evidence that hardware prefetching can significantly improve the performance of virtual memory systems since it is a fully legacy-preserving hardware technique that relies only on the memory access patterns of the applications, it does not disrupt existing the existing software and hardware stacks, and is independent of the system state, without requiring any OS involvement. In this direction, we introduce two novel TLB prefetching schemes that reduce the TLB miss rates of contemporary memory-intensive and large code footprint applications, and a novel module that enhances the performance of hardware cache prefetchers operating in the physical address space by exploiting the address translation metadata available at the $\mu$architectural levels.

To improve the performance of address translation associated with data accesses, we propose a unified solution that consists of the *Sampling-Based Free TLB Prefetching (SBFP)* scheme and the *Agile TLB Prefetcher (ATP)*. SBFP is a μarchitectural scheme that exploits the locality in the last level of the radix tree page table to improve the performance and reduce the memory footprint of TLB prefetching. ATP is a TLB prefetcher that combines three low-cost engines by using adaptive selection logic while disabling TLB prefetching during phases that is not useful.

Next, we provide the first μarchitectural work to characterize the instruction TLB behavior of industrial server applications while providing evidence that instruction address translation is an emerging performance bottleneck in servers and datacenters. To improve the performance of address translation associated with instruction accesses for big code server applications, we propose *Morrigan*, the first ever hardware TLB prefetcher for instruction accesses. Morrigan consists of two complementary prefetch engines: (i) the Irregular Instruction TLB Prefetcher (IRIP) which is an ensemble of hardware Markov prefetchers that dynamically build variable length Markov chains out of the instruction TLB miss stream while using a novel frequency-based replacement policy, and (ii) the Small Delta Prefetcher (SDP) which is a simple sequential TLB prefetcher.

The last contribution of this thesis targets to improve the efficacy of cache prefetchers operating in the physical address space by exploiting modern prevalence and support for large pages. We propose the *Page-size Propagation Module (PPM)*, a μarchitectural scheme that efficiently propagates the page size information to the lower-level cache prefetchers and enables safe prefetching beyond 4KB physical page boundaries when the accessed block resides in a large page, at the cost of augmenting the MSHRs of the first-level caches with one bit. In addition, we design a composite prefetching module that consists of two prefetch engines that both exploit PPM but drive prefetching decisions assuming different page sizes. Finally, this composite design uses adaptive selection logic to dynamically switch between the two page size aware prefetchers and is transparent to which prefetcher is used.

The proposals of this dissertation constitute practical solutions to real-world bottlenecks of HPC and cloud systems since they incur minimal storage and logic overheads without disrupting any software or hardware stack while being compatible with legacy systems. Therefore, they have great potential to influence future industrial designs.

## 6.1 Additional Future Work

Sections 3.11, 4.9, and 5.8, present potentially interesting future research directions for each contribution of this dissertation. Apart from them, a potentially interesting research direction is to examine the applicability of the proposals of this doctoral thesis on GPU systems where address translation is also a major performance bottleneck [223, 223, 254, 254, 255] despite the existence of sophisticated architectural support aimed at reducing the overheads of frequent page walks.

## 6.2 Additional Acknowledgements

This thesis has been partially supported by the Generalitat de Catalunya (contract 2017-SGR-1414), by the European Union Horizon 2020 research and innovation program under grant agreement No 955606 (DEEP-SEA EU project), by the grant RYC-2017-23269 funded by MCIN/AEI/10.13039/501100011033 and ESF 'Investing in your future', and by the Spanish Ministry of Economy, Industry and Competitiveness and the European Social Fund under the FPI fellowship No. PRE2018-087046, the Juan de la Cierva Formacion fellowship No. FJCI-2016-30984, the Ramon y Cajal fellowship No. RYC-2017-23269, and the PID2019-107255GB project.

# 7 Publications

## 7.1 Conference Publications

The research output of the present dissertation led to the following conference publications where the author is the leading investigator:

**Publication 1**

**Georgios Vavouliotis**, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, Marc Casas, **Exploiting Page Table Locality for Agile TLB Prefetching**. In Proceedings of the 48th International Symposium on Computer Architecture, **ISCA 2021**, doi: 10.1109/ISCA52012.2021.00016

*This work has also been presented as a **poster** [280] at the 49th International Symposium on Computer Architecture, **ISCA 2022**.*

**Publication 2**

**Georgios Vavouliotis**, Lluc Alvarez, Boris Grot, Daniel A. Jiménez, Marc Casas, **Morrigan: A Composite Instruction TLB Prefetcher**. In Proceedings of the 54th International Symposium on Microarchitecture, **MICRO 2021**, doi: 10.1145/3466752.3480049.

**Publication 3**

**Georgios Vavouliotis**, Gino Chancon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, Marc Casas, **Page Size Aware Cache Prefetching**. In Proceedings of the 55th International Symposium on Microarchitecture, **MICRO 2022**, doi: 10.1109/MICRO56248.2022.00070

*This work has also participated in the **2021 ACM Student Research Competition (SRC)** [282] held at the 54th International Symposium on Microarchitecture, MICRO 2021.*

*This work has also been presented as a **poster** [281] at the **2021 ACM Summer School on HPC Computer Architectures for AI and Dedicated Applications.** [Best Poster Award]*

## 7.2 Other Publications

The research output of the present dissertation led to the following workshop/competition publications where the author is the leading investigator:

**Publication 4**

**Georgios Vavouliotis**, Lluc Alvarez, Daniel A. Jiménez, Marc Casas, **Cost-Effective Instruction TLB Prefetching**. In Proceedings of the Second Young Architect Workshop (YArch 2020). In conjunction with The 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

**Publication 5**

**Georgios Vavouliotis**, Lluc Alvarez, Marc Casas, **Pushing the Envelope on Free TLB Prefetching**. In Proceedings of the 8th BSC International Doctoral Symposium, BSC 2021.

Furthermore, the author participated in another research work carried out by members of the same research group. The content of this publication has not been included in this thesis since the author contributed as a collaborator instead of leading the investigation and the work was under submission by the time of the thesis submission.

**Publication 6**

Alexander V. Jamet, **Georgios Vavouliotis**, Lluc Alvarez, Daniel A. Jiménez, Marc Casas, **A Two-Step Neural Approach Combining Off-Chip Prediction with Adaptive Prefetch Filtering**. [under submission]

Finally, the author has also researched memory management during his internship at Huawei Research Center Zurich, leading to a conference publication. This work is not part of the present dissertation.

**Publication 7**

Maria Carpen-Amarie, **Georgios Vavouliotis**, Konstantinos Tovletoglou, Boris Grot, Rene Mueller, **Concurrent GCs and Modern Java Workloads: A Cache Perspective**. In Proceedings of the 2023 International Symposium on Memory Management, **ISMM 2023**.

# 8 Research Vision

Modern processor designs leverage various $\mu$architectural predictors and prefetchers to bridge the processor-memory performance gap. In practice, fundamental computer architecture concepts like ILP, out-of-order execution, and speculative execution heavily depend on the effectiveness of $\mu$architectural prefetchers and predictors.

Within the realm of computing, my research aims at identifying and exploiting the predictability of programs to design $\mu$architectural prefetching and prediction mechanisms for the cache and the TLB hierarchy, improving cache/TLB management for emerging applications with large data and code footprints, leveraging machine learning algorithms to design intelligent $\mu$architectural components, and re-thinking $\mu$architectural designs for server and data center applications.

In my perspective, a new golden age for computer architecture is rising as non-conventional and groundbreaking approaches are needed to overcome the barrier that is placed by the end of Dennard Scaling and the slowing of Moore's law [215].

My research vision for the next years is to build **intelligent $\mu$architectures**. In this direction, I aim to (i) conduct software-assisted $\mu$architectural research by exploiting infor-

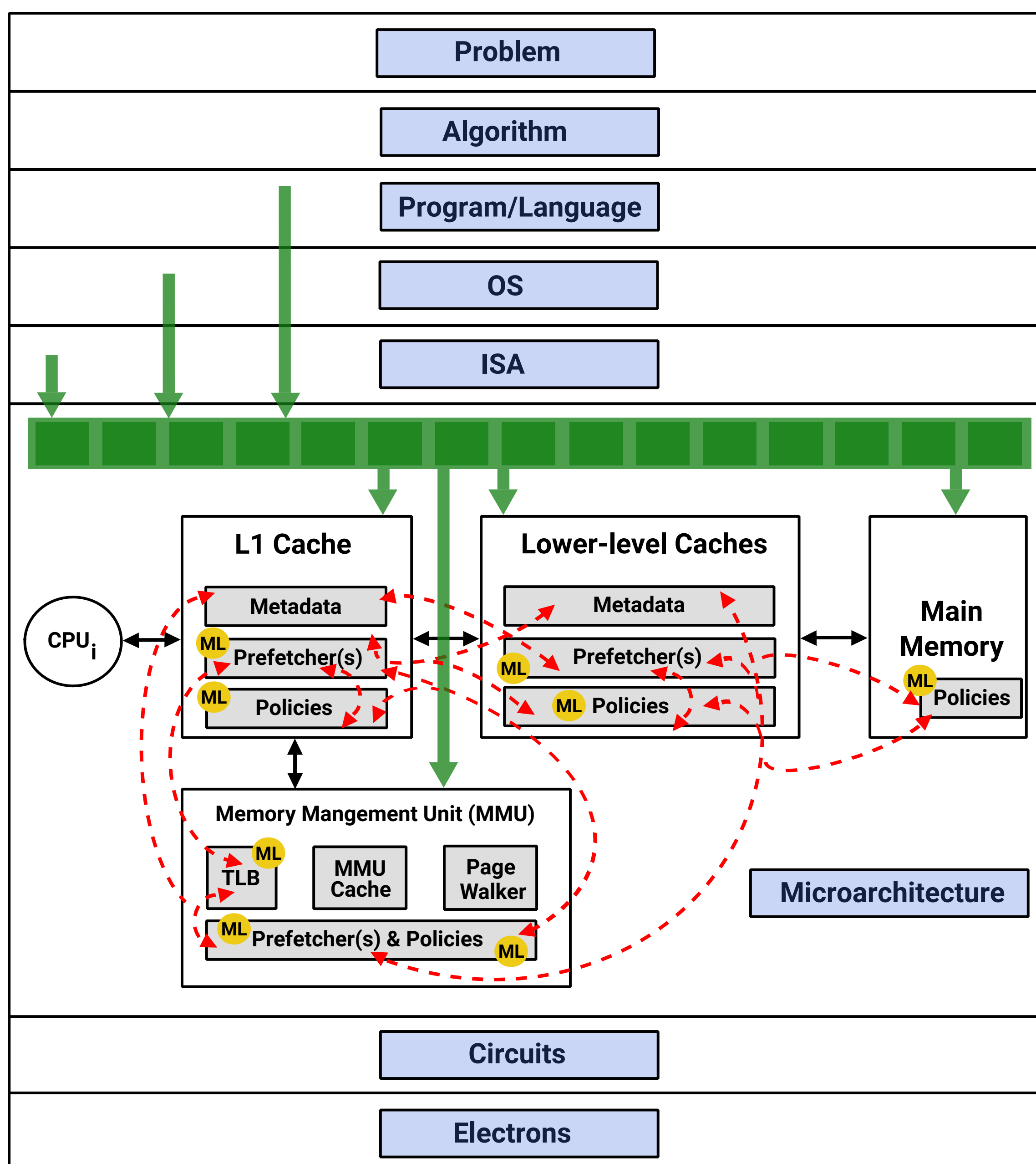


Figure 8.1: Research vision illustrated with a cartoon.

mation available in different layers of the transformation hierarchy to push the envelope on $\mu$architectural research, given real-world technology constraints, and (ii) build smart $\mu$architectural prefetchers and predictors that internally leverage machine learning algorithms. Figure 8.1 illustrates my research vision for the next years.

## 8.1 Software-Assisted $\mu$architectural Designs

The core idea is to pass software-sourced hints to the $\mu$architecture, through the ISA, that can improve the efficacy of the $\mu$architectural components. Hardware prefetching is one possible application of this idea (not the only one); the software encodes information about the nature of the memory access patterns and transmits them to the hardware prefetchers, which take them into account and adjust their prefetching strategy accordingly.

Another particularly appealing research direction is data and instruction cache prefetching via hybridization techniques. I envision the design of multiple low-cost hardware prefetchers with low prefetching scope and high accuracy. To make this idea a success, a sophisticated and highly accurate scheme that identifies the nature of the memory accesses and accordingly enables the most appropriate prefetcher is required. This scheme could be either a pure hardware predictor or a low-cost $\mu$architectural component that leverages software-sourced encoded information about the nature of the access patterns.

## 8.2 ML for $\mu$architectural Prediction and Prefetching

Machine learning has provided great performance and accuracy gains in various $\mu$architectural domains. The most prevalent examples are branch prediction [156] and cache replacement policies [157]. Moreover, recent works [76, 252] highlight that ML-based hardware prefetchers have the potential to provide great benefits. The core idea behind my vision is that well-established ML algorithms could be used to design intelligent and practical $\mu$architectural prefetchers, predictors, and replacement policies for the TLB and cache hierarchy capable of providing great performance, coverage, and accuracy enhancements.

## 8.3 Other Domains

Apart from the domains presented above, I am particularly interested in conducting research on the following computer architecture areas: (i) interaction between TLB management and branch prediction, (ii) TLB management for native and virtualized environments, (iii) address translation for hybrid memory systems (*e.g.,* DRAM-NVMM), and (iv) serverless computing, among others.

## 8.4 Industry, Academia, and Teaching

My research vision also involves both industrial and academic collaborations since it aims at effectively tackling the upcoming technological challenges in a practical and impactful manner. Apart from that, my future plans include teaching computer architecture-related courses as well as advising students since I firmly believe that tutoring is the best way to master a topic.

# Abbreviations

**ASP:** Arbitrary Stride Prefetcher
**ATP:** Agile TLB Prefetcher
**BD:** Big Data
**BOP:** Best Offset Prefetcher
**CC:** CPU Cycles
**CPU:** Centra Processor Unit
**CVP-1:** 1st Contest on Value Prediction
**DDR4:** Double Data Rate 4
**DP:** Distance Prefetcher
**FDT:** Free Distance Table
**FPB:** Fake Prefetch Buffer
**FSM:** Finite State Machine
**GPU:** Graphics Processing Unit
**H2P:** H2 Prefetcher
**HPC:** High Performance Computing
**ILP:** Instruction Level Parallelism
**IP:** Instruction Pointer
**IPC:** Instructions Per Cycle
**IPCP:** Instruction Pointer Classifier Prefetcher
**IPC-1:** 1st Instruction Prefetching Championship
**IRIP:** Irregular Instruction TLB Prefetcher
**L1d/L1D:** L1 Data Cache
**LFU:** Least Frequently Used
**L1i/L1I:** L1 Instruction Cache

**L2C:** L2 Cache
**LLC:** Last Level Cache
**LRU:** Least Recently Used
**MASP:** Modified Arbitrary Stride Prefetcher
**ML:** Machine Learning
**MLP:** Memory Level Parallelism
**MMU:** Memory Management Unit
**MP:** Markov Prefetcher
**MPKI:** Misses Per Kilo Instructions
**MSHR:** Miss Status Holding Register
**NL:** Next Line
**PB:** Prefetch Buffer
**PC:** Program Counter
**PD:** Page Directory
**PDP:** Page Directory Pointer
**PIPT:** Physically Indexed Physically Tagged
**PIVT:** Physically Indexed Virtually Tagged
**PML4:** Page Map Level 4
**PPF:** Perceptron-based Prefetch Filtering
**PPM:** Page-size Propagation Module
**PSA:** Page Size Aware
**PSA-SD:** Page Size Aware with Set Dueling
**PSC:** Page Structure Cache
**PT:** Page Table
**PTE:** Page Table Entry
**PTW:** Page Table Walker
**QMM:** Qualcomm
**SBFP:** Sampling-Based Free TLB Prefetching
**SDP:** Small Delta Prefetcher
**SP:** Sequential Prefetcher
**STP:** Stride Prefetcher
**SPP:** Signature Path Prefetcher
**TLB:** Translation Lookaside Buffer

**VIPT:** Virtually Indexed Physically Tagged
**VIVT:** Virtually Indexed Virtually Tagged
**VLDP:** Variable Length Delta Prefetcher
**VPN:** Virtual Page Number

# Bibliography


[1] Intel 5-Level Paging and 5-Level EPT, https://mobt3ath.com/uplode/books/book-51381.pdf. Cited on page: 5, 32, 85, 138

[2] perl-Net-Amazon-EC2, https://ppisar.fedorapeople.org/perl_rebuild/scratch/latest/packages/perl-Net-Amazon-EC2/hw_info.log, Accessed: 05-9-2022. Cited on page: 141

[3] perl-Net-Amazon-S3, https://ppisar.fedorapeople.org/perl_rebuild/scratch/latest/packages/perl-Net-Amazon-S3/hw_info.log, Accessed: 05-9-2022. Cited on page: 141

[4] perl-Net-Amazon, https://ppisar.fedorapeople.org/perl_rebuild/scratch/latest/packages/perl-Net-Amazon/hw_info.log, Accessed: 05-9-2022. Cited on page: 141

[5] AMD64 Architecture Programmer Manual (Volume 2), https://www.amd.com/system/files/TechDocs/24593.pdf. Cited on page: 6, 17, 138

[6] AMD EPYC™ 7003 Processors, https://www.amd.com/en/server-docs/software-optimization-guide-for-amd-epyc-7003-processors-zip-format. Cited on page: 29

[7] AMD-V™ Nested Paging, http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf. Cited on page: 35, 161

[8] Database Tuning on Linux OS: Reference Guide for AMD EPYC™ 7002 Series Processors, https://developer.amd.com/wp-content/resources/56783_1.0.pdf. Cited on page: 35, 161

[9] Arm Architecture Reference Manual for A-profile Architecture, https://developer.arm.com/documentation/ddi0487/latest. Cited on page: 6, 17, 138

[10] Virtual Memory Support, https://developer.arm.com/documentation/ddi0406/b/Appendices/ARMv4-and-ARMv5-Differences/System-level-memory-model/Virtual-memory-support?lang=en. Cited on page: 6, 35, 161

[11] Public Clouds And Vulnerable CPUs: Are We Secure?, https://archive.fosdem.org/2020/schedule/event/vai_pubic_clouds_and_vulnerable_cpus/attachments/slides/3650/export/events/attachments/vai_pubic_clouds_and_vulnerable_cpus/slides/3650/FOSDEM2020_vkuznets.pdf. Cited on page: 141

[12] A Principled Technologies Report: Hands-on Testing. Real-world Results, https://www.principledtechnologies.com/Intel/Xeon-8272CL-Microsoft-Azure-WordPress-science-0920.pdf. Cited on page: 141

[13] ChampSim, https://crc2.ece.tamu.edu/, Accessed: 05-9-2022. Cited on page: 67, 85, 120, 139, 169, 181

[14] Coffee Lake - Microarchitectures - Intel, https://en.wikichip.org/wiki/intel/microarchitectures/coffee_lake, Accessed: 05-9-2022. Cited on page: 119

[15] Arm Architecture Reference Manual for A-profile Architecture – Data Prefetching, https://developer.arm.com/documentation/100442/0100/functional-description/level-1-memory-system/data-prefetching. Cited on page: 28

[16] Arm Architecture Reference Manual – Load/Store Hardware Prefetcher, https://developer.arm.com/documentation/100095/0002/level-1-memory-system/l1-data-memory-system/load-store-hardware-prefetcher. Cited on page: 28

[17] CVE-2017-15127, https://nvd.nist.gov/vuln/detail/CVE-2017-15127, Accessed: 05-9-2022. Cited on page: 176

[18] CVE-2021-4002, https://nvd.nist.gov/vuln/detail/CVE-2021-4002, Accessed: 05-9-2022. Cited on page: 176

[19] Championship Value Prediction (CVP), https://www.microarch.org/cvp1/, Accessed: 05-9-2022. Cited on page: 63, 67, 88, 116, 117, 119, 141, 163, 183

[20] Decision Tree Learning, https://en.wikipedia.org/wiki/Decision_tree_learning, Accessed: 05-9-2022. Cited on page: 11, 78

[21] The Locality Principle, http://denninginstitute.com/pjd/PUBS/locality_2006.pdf. Cited on page: 121

[22] Before Memory Was Virtual, http://denninginstitute.com/pjd/PUBS/bvm.pdf. Cited on page: 121

[23] Disclosure of Hardware Prefetcher Control on Some Intel Processors, https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html, Accessed: 05-9-2022. Cited on page: 3, 21, 28

[24] D-JOLT: Distant Jolt Prefetcher, https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/D-JOLT.pdf. Cited on page: 127

[25] AMD EPYC 7702P, https://en.wikichip.org/wiki/amd/epyc/7702p, Accessed: 05-9-2022. Cited on page: 182

[26] Partition Allocation in Memory Management, https://www.geeksforgeeks.org/partition-allocation-methods-in-memory-management/, Accessed: 05-9-2022. Cited on page: 50

[27] Difference Between Internal Fragmentation and External Fragmentation, https://www.geeksforgeeks.org/difference-between-internal-and-external-fragmentation/, Accessed: 05-9-2022. Cited on page: 50

[28] Haswell - Microarchitectures - Intel, https://en.wikichip.org/wiki/intel/microarchitectures/haswell_(client), Accessed: 05-9-2022. Cited on page: 119

[29] Intel Ice Lake, https://www.7-cpu.com/cpu/Ice_Lake.html, Accessed: 05-9-2022. Cited on page: xxiii, 19, 20

[30] Intel® 64 and IA-32 Architectures Software Developer Manuals, https://software.intel.com/en-us/articles/intel-sdm, Accessed: 05-9-2022. Cited on page: 17, 35, 138, 161

[31] Intel®64 and IA-32 Architectures Optimization Reference Manual, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf. Cited on page: 5, 6, 37, 161

[32] The 1st Instruction Prefetching Championship, https://research.ece.ncsu.edu/ipc/, Accessed: 05-9-2022. Cited on page: 115, 117, 119, 127, 139, 141, 151, 155

[33] Kernel Address Space Layout Randomization, https://lwn.net/Articles/569635/, Accessed: 05-9-2022. Cited on page: 49, 161

[34] iTLB Multihit, https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/multihit.html. Cited on page: 141

[35] Page-Collect, https://github.com/cslab-ntua/contiguity-isca2020, Accessed: 05-2-2021. Cited on page: 168, 182

[36] Inter-core Cooperative TLB Prefetchers, https://patents.google.com/patent/US8880844B1/en. Cited on page: 52, 150

[37] perf: Linux Profiling with Performance Counters., https://perf.wiki.kernel.org, Accessed: 05-9-2022. Cited on page: 118

[38] The RISC-V Instruction Set Manual, https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf. Cited on page: 17, 138

[39] AMD Ryzen Threadripper 3990X, https://en.wikichip.org/wiki/amd/ryzen_threadripper/3990x, Accessed: 05-9-2022. Cited on page: 182

[40] Sandy Bridge - Microarchitectures - Intel, https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client), Accessed: 05-9-2022. Cited on page: 119

[41] SPEC CPU 2006, https://www.spec.org/cpu2006/, Accessed: 05-9-2022. Cited on page: 119, 142

[42] SPEC CPU 2017, https://www.spec.org/cpu2017/, Accessed: 05-9-2022. Cited on page: xxix, 63, 67, 88, 119, 142, 163, 167, 168, 182, 183

[43] Transparent Huge Pages, http://lwn.net/Articles/423584/, Accessed: 05-9-2022. Cited on page: 15, 35, 110, 119, 161, 166, 168, 182, 200

[44] Intel Xeon Gold, https://en.wikichip.org/wiki/intel/xeon_gold/6258r, Accessed: 05-9-2022. Cited on page: 162, 182

[45] XSBench, https://github.com/ANL-CESAR/XSBench, Accessed: 05-9-2022. Cited on page: 63, 67, 88

[46] Huge Pages/libhugetlbfs, https://lwn.net/Articles/374424/, Accessed: 05-9-2022. Cited on page: 119, 140, 166, 176, 182, 200

[47] Using Intel VTune Amplifier XE, https://www.vi-hps.org/cms/upload/material/tw11/VTune_Amplifier_XE_Overview.pdf. Cited on page: 121

[48] Abishek Bhattacharjee, Advanced Concepts on Address Translation, Appendix L in "Computer Architecture: A Quantitative Approach" by Hennessy and Patterson, http://www.cs.yale.edu/homes/abhishek/abhishek-appendix-l.pdf. Cited on page: xxv, 44, 85, 111, 119, 137, 139, 140, 150, 161, 167

[49] Jeongseob Ahn, Seongwook Jin, and Jaehyuk Huh. Revisiting Hardware-Assisted Page Walks for Virtualized Systems. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 476–487, 2012. DOI: 10.1109/ISCA.2012.6237041. Cited on page: 31

[50] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of Optimal Page Replacement. *Journal of the ACM*, 18(1):80–93, 1971. DOI: 10.1145/321623.321632. Cited on page: 50

[51] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *Proceedings of the 44th International Symposium on Computer Architecture*, ISCA '17, pages 457–468, 2017. DOI: 10.1145/3079856.3080209. Cited on page: 10, 13, 109, 114, 154

[52] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the 2020 47th International Symposium on Computer Architecture*, ISCA '20, pages 515–528, 2020. DOI: 10.1109/ISCA45697.2020.00050. Cited on page: 41, 46, 61, 110, 161, 168

[53] Nadav Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Technical Conference*, USENIX ATC '17, page 27–39, 2017. Cited on page: 31, 137, 177

[54] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967. DOI: 10.1147/rd.111.0008. Cited on page: 27

[55] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. *SIGARCH Computer Architecture News*, 19(2):108–120, 1991. DOI: 10.1145/106975.106985. Cited on page: 4, 5, 10, 31, 41, 46, 61, 161

[56] Jean Araujo, Rubens Matos, Paulo Maciel, Rivalino Matias, and Ibrahim Beicker. Experimental Evaluation of Software Aging Effects on the Eucalyptus Cloud Computing Infrastructure. In *Proceedings of the Middleware 2011 Industry Track Workshop*, Middleware '11, 2011. DOI: 10.1145/2090181.2090185. Cited on page: 140

[57] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *Proceedings*

*of the 26th International Conference on Parallel Architectures and Compilation Techniques*, PACT '17, pages 273–287, 2017. DOI: 10.1109/PACT.2017.38. Cited on page: 31, 137, 177

[58] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. Memory Hierarchy for Web Search. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture*, HPCA '18, pages 643–656, 2018. DOI: 10.1109/HPCA.2018.00061. Cited on page: 4, 10, 31, 41, 46, 61, 161

[59] Vlastimil Babka and Petr Tůma. Investigating Cache Parameters of X86 Family Processors. page 77–96, 2009. DOI: 10.1007/978-3-540-93799-9_5. Cited on page: 28

[60] Jean-Loup Baer and Tien-Fu Chen. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995. DOI: 10.1109/12.381947. Cited on page: 52, 54, 86

[61] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad. Bingo Spatial Data Prefetcher. In *Proceedings of the 25th International Symposium on High Performance Computer Architecture*, HPCA '19, pages 399–411, 2019. DOI: 10.1109/HPCA.2019.00053. Cited on page: 2, 3, 14, 26, 27, 160

[62] Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Domino Temporal Data Prefetcher. In *Proceedings of the 24th International Symposium on High Performance Computer Architecture*, HPCA '18, pages 131–142, 2018. DOI: 10.1109/HPCA.2018.00021. Cited on page: 2, 3, 14, 26, 27, 160

[63] Chinnakrishnan Ballapuram, Kiran Puttaswamy, Gabriel H. Loh, and Hsien-Hsin S. Lee. Entropy-Based Low Power Data TLB Design. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 304–311, 2006. DOI: 10.1145/1176760.1176797. Cited on page: 31, 46

[64] Chinnakrishnan S. Ballapuram, Hsien-Hsin S. Lee, and Milos Prvulovic. Synonymous Address Compaction for Energy Reduction in Data TLB. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, page 357–362, 2005. DOI: 10.1145/1077603.1077689. Cited on page: 31, 46

[65] T. W. Barr, A. L. Cox, and S. Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA '11, 2011. Cited on page: 61, 110

[66] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don'T Walk (the Page Table). In *Proceedings of the 37th International Symposium on Computer Architecture*, ISCA '10, pages 48–59, 2010. DOI: 10.1145/1815961.1815970. Cited on page: 4, 7, 10, 31, 48, 61, 109, 154, 161

[67] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *Proceedings of the 38th International Symposium on Computer Architecture*, ISCA '11, pages 307–318, 2011. DOI: 10.1145/2000064.2000101. Cited on page: 6, 47, 154

[68] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. *Synthesis Lectures on Computer Architecture*, 13(3), 2018. DOI: 10.2200/S00874ED3V01Y201809CAC046. Cited on page: 117

[69] Arkaprava Basu. *Revisiting Virtual Memory*. PhD thesis, University of Wisconsin at Madison, 2013. Cited on page: 7, 31

[70] Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Reducing Memory Reference Energy with Opportunistic Virtual Caching. In *Proceedings of the 39th International Symposium on Computer Architecture*, ISCA '12, pages 297–308, 2012. DOI: 10.1109/ISCA.2012.6237026. Cited on page: 45, 46

[71] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th International Symposium on Computer Architecture*, ISCA '13, pages 237–248, 2013. DOI: 10.1145/2485922.2485943. Cited on page: 4, 5, 6, 10, 13, 31, 41, 46, 47, 61, 110, 161

[72] M. Bazm, M. Lacoste, M. Südholt, and J. Menaud. Side-channels Beyond the Cloud Edge: New Isolation Threats and Solutions. In *Proceedings of the 1st Cyber Security in Networking Conference*, CSNet '17, pages 1–8, 2017. DOI: 10.1109/CSNET.2017.8241986. Cited on page: 141

[73] Mohammad-Mahdi Bazm, Marc Lacoste, Mario Südholt, and Jean-Marc Menaud. Side Channels in the Cloud: Isolation Challenges, Attacks, and Countermeasures. working paper or preprint, 2017. PDF: https://hal.inria.fr/hal-01591808/file/sca_survey.pdf. Cited on page: 141

[74] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP Benchmark Suite. *CoRR*, abs/1508.03619, 2015. Cited on page: xxix, xxxiii, 63, 67, 88, 163, 167, 168, 182, 183

[75] Rahul Bera, Anant V. Nori, Onur Mutlu, and Sreenivas Subramoney. DSPatch: Dual Spatial Pattern Prefetcher. In *Proceedings of the 52nd International Symposium on Microarchitecture*, MICRO '19, page 531–544, 2019. DOI: 10.1145/3352460.3358325. Cited on page: 2, 3, 14, 26, 27, 160, 169, 170

[76] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *Proceedings of the 54th International Symposium on Microarchitecture*, MICRO '21, page 1121–1137, 2021. DOI: 10.1145/3466752.3480114. Cited on page: 2, 3, 26, 112, 160, 182, 211

[77] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGARCH Comput. Archit. News*, 28(5):117–128, 2000. DOI: 10.1145/378995.379232. Cited on page: 31, 50

[78] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering Custom Memory Allocation. In *Proceedings of the 17th SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, page 1–12, 2002. DOI: 10.1145/582419.582421. Cited on page: 31, 50

[79] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, pages 26–35, 2008. DOI: 10.1145/1346281.1346286. Cited on page: 111

[80] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. Perceptron-Based Prefetch Filtering. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 1–13, 2019. DOI: 10.1145/3307650.3322207. Cited on page: xxix, 2, 3, 16, 26, 112, 160, 162, 164, 169, 170, 181, 184

[81] Abhishek Bhattacharjee. *Thread Criticality and TLB Enhancement Techniques for Chip Multiprocessors*. PhD thesis, Princeton University, 2010. Cited on page: 31

[82] Abhishek Bhattacharjee. Large-reach Memory Management Unit Caches. In *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO '13, pages 383–394, 2013. DOI: 10.1145/2540708.2540741. Cited on page: 4, 5, 7, 10, 31, 41, 46, 48, 61, 109, 113, 154, 161

[83] Abhishek Bhattacharjee. Translation-Triggered Prefetching. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 63–76, 2017. DOI: 10.1145/3037697.3037705. Cited on page: 43

[84] Abhishek Bhattacharjee. Preserving Virtual Memory by Mitigating the Address Translation Wall. *IEEE Micro*, 37(5):6–10, 2017. DOI: 10.1109/MM.2017.3711640. Cited on page: 4, 5, 10, 31, 46

[85] Abhishek Bhattacharjee and Margaret Martonosi. Inter-core Cooperative TLB for Chip Multiprocessors. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '10, pages 359–370, 2010. DOI: 10.1145/1736020.1736060. Cited on page: 10, 47, 52, 57, 62, 105, 109, 113, 150

[86] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared Last-level TLBs for Chip Multiprocessors. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 62–63, 2011. Cited on page: 6, 11, 13, 43, 47, 62, 64, 105, 109, 113, 154

[87] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool Publishers, 2017. Cited on page: xxiv, 4, 30, 31, 32, 34, 35, 36, 37, 44, 49, 50, 161

[88] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '89, page 113–122, 1989. DOI: 10.1145/70082.68193. Cited on page: 137, 177

[89] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, 2006. DOI: 10.1145/1167473.1167488. Cited on page: xxvii, 118

[90] Ulrik Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001. Cited on page: 88

[91] Richard W. Carr and John L. Hennessy. WSCLOCK—a Simple and Effective Algorithm for Virtual Memory Management. In *Proceedings of the 8th Symposium on Operating Systems Principles*, SOSP '81, page 87–95, 1981. DOI: 10.1145/800216.806596. Cited on page: 35, 50, 161

[92] Richard P. Case and Andris Padegs. Architecture of the IBM System/370. *Commun. ACM*, 21(1):73–96, 1978. DOI: 10.1145/359327.359337. Cited on page: 27

[93] Xiaotao Chang, Hubertus Franke, Yi Ge, Tao Liu, Kun Wang, Jimi Xenidis, Fei Chen, and Yu Zhang. Improving Virtualization in the Presence of Software Managed Translation Lookaside Buffers. In *Proceedings of the 40th International Symposium on Computer Architecture*, ISCA '13, page 120–129, 2013. DOI: 10.1145/2485922.2485933. Cited on page: 47

[94] Yen-Jen Chang and Mao-Feng Lan. Two New Techniques Integrated for Energy-Efficient TLB Design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(1):13–23, 2007. DOI: 10.1109/TVLSI.2006.887813. Cited on page: 46

[95] J. Bradley Chen, Anita Borg, and Norman P. Jouppi. A Simulation Based Study of TLB Performance. In *Proceedings of the 19th International Symposium on Computer*

*Architecture*, ISCA '92, page 114–123, 1992. DOI: 10.1145/139669.139708. Cited on page: 31

[96] Yun Chen, Lingfeng Pei, and Trevor E. Carlson. Leaking Control Flow Information via the Hardware Prefetcher. *CoRR*, abs/2109.00474, 2021. Cited on page: 59, 161

[97] Jin-Hyuck Choi, Jung-Hoon Lee, Seh-Woong Jeong, Shin-Dug Kim, and C. Weems. A Low Power TLB Structure for Embedded Systems. *IEEE Computer Architecture Letters*, 1(1):3–3, 2002. DOI: 10.1109/L-CA.2002.1. Cited on page: 31, 46

[98] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Trans. Comput. Syst.*, 3(1):31–62, 1985. DOI: 10.1145/214451.214455. Cited on page: 31

[99] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 Translation Buffer: Simulation and Measurement. *ACM Transactions on Computer Systems*, 3 (1):31–62, 1985. DOI: 10.1145/214451.214455. Cited on page: 4, 10, 31, 61

[100] J. Collins, S. Sair, B. Calder, and D.M. Tullsen. Pointer Cache Assisted Prefetching. In *Proceedings of the 35th International Symposium on Microarchitecture*, MICRO '02, pages 62–73, 2002. DOI: 10.1109/MICRO.2002.1176239. Cited on page: 29

[101] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A Stateless, Content-Directed Data Prefetching Mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '02, page 279–290, 2002. DOI: 10.1145/605397.605427. Cited on page: 29

[102] Guilherme Cox. *Improving and Complementing Virtual Memory Using Hardware Techniques*. PhD thesis, Rutgers University, 2018. Cited on page: 31

[103] Guilherme Cox and Abhishek Bhattacharjee. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 435–448, 2017. DOI: 10.1145/3037697.3037704. Cited on page: 10, 31, 110, 114

[104] Ryan R. Curtin, James R. Cline, N. P. Slagle, William B. March, Parikshit Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A Scalable C++ Machine Learning Library. *J. Mach. Learn. Res.*, 14(1):801–805, 2013. Cited on page: 163, 183

[105] F. Dahlgren and P. Stenstrom. Effectiveness of Hardware-based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. In *Proceedings of the 1st International Symposium on High Performance Computer Architecture*, HPCA '95, pages 68–77, 1995. DOI: 10.1109/HPCA.1995.386554. Cited on page: 2, 29, 54

[106] Peter J. Denning. Virtual Memory. *ACM Comput. Surv.*, 2(3):153–189, 1970. DOI: 10.1145/356571.356573. Cited on page: 50, 121

[107] Xiaowan Dong, Sandhya Dwarkadas, and Alan L. Cox. Shared address translation revisited. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16, 2016. DOI: 10.1145/2901318.2901327. Cited on page: 114, 155

[108] Kshitij Doshi and Jantz Tran. Using Hugetlbfs for Mapping Application Text Regions. 2006. Cited on page: 141, 155

[109] Cort Dougan, Paul Mackerras, and Victor Yodaiken. Optimizing the Idle Task and Other MMU Tricks. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, page 229–237, 1999. Cited on page: 31

[110] Yu Du, Miao Zhou, Bruce R. Childers, Daniel Mossé, and Rami Melhem. Supporting Superpages in Non-Contiguous Physical Memory. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA '15, pages 223–234, 2015. DOI: 10.1109/HPCA.2015.7056035. Cited on page: 110

[111] Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, HPCA '09, pages 7–17, 2009. DOI: 10.1109/HPCA.2009.4798232. Cited on page: 29

[112] Hussein Elnawawy, Rangeen Basu Roy Chowdhury, Amro Awad, and Gregory T. Byrd. Diligent TLBs: A Mechanism for Exploiting Heterogeneity in TLB Miss Be-

havior. In *Proceedings of the International Conference on Supercomputing*, ICS '19, page 195–205, 2019. DOI: 10.1145/3330345.3330363. Cited on page: 155

[113] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *Proceedings of the 49th International Symposium on Microarchitecture*, MICRO '16, 2016. Cited on page: 141

[114] Babak Falsafi and Thomas F. Wenisch. *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers, 2014. Cited on page: 1, 2, 18, 26

[115] Dongrui Fan, Zhimin Tang, Hailin Huang, and Guang R. Gao. An Energy Efficient TLB Design Methodology. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, page 351–356, 2005. DOI: 10.1145/1077603.1077688. Cited on page: 31, 46

[116] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal Instruction Fetch Streaming. In *Proceedings of the 41st International Symposium on Microarchitecture*, MICRO '08, pages 1–10, 2008. DOI: 10.1109/MICRO.2008.4771774. Cited on page: 2, 3, 26, 27

[117] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 37–48, 2012. DOI: 10.1145/2150976.2150982. Cited on page: 4, 10, 31, 41, 46, 61, 115, 118, 155, 161, 163, 183

[118] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proceedings of the 20th Symposium on Theoretical Aspects of Computer Science*, STACS '03, page 271–282, 2003. Cited on page: 85, 110, 138, 154

[119] Narayanan Ganapathy and Curt Schimmel. General Purpose Operating System Support for Multiple Page Sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 8–8, 1998. Cited on page: 110

[120] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *Proceedings of the 47th International Symposium on Microarchitecture*, MICRO '14, pages 178–189, 2014. DOI: 10.1109/MICRO.2014.37. Cited on page: 6, 47, 111

[121] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 707–718, 2016. DOI: 10.1109/ISCA.2016.67. Cited on page: 31

[122] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '14, page 231–242, 2014. Cited on page: 140

[123] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium*, USENIX Security '12, pages 475–490, 2012. Cited on page: 31, 35, 49, 161

[124] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim, The championship simulator: Architectural simulation for education and competition. Cited on page: 67, 85, 120, 139, 169, 181

[125] Mel Gorman and Patrick Healy. Performance Characteristics of Explicit Superpage Support. In *Proceedings of the 37th International Conference on Computer Architecture*, ISCA '10, page 293–310, 2010. DOI: 10.1007/978-3-642-24322-6_24. Cited on page: 35

[126] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya. Evolution of the Samsung Exynos CPU Microarchitecture. In *Proceedings of the 47th International Symposium on Computer Architecture*, ISCA '20, pages 40–51, 2020. DOI: 10.1109/ISCA45697.2020.00015. Cited on page: 3, 21, 27, 28, 29, 169

[127] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A Case for Unlimited Watchpoints. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 159–172, 2012. DOI: 10.1145/2150976.2150994. Cited on page: 31

[128] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, 2016. DOI: 10.1145/2976749.2978356. Cited on page: 14, 59, 161

[129] Fei Guo, Seongbeom Kim, Yury Baskakov, and Ishan Banerjee. Proactively Breaking Large Pages to Improve Memory Overcommitment Performance in VMware ESXi. In *Proceedings of the 11th SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, page 39–51, 2015. DOI: 10.1145/2731186.2731187. Cited on page: 35, 161

[130] F. Guvenilir and Y. N. Patt. Tailored Page Sizes. In *Proceedings of the 47th International Symposium on Computer Architecture*, ISCA '20, 2020. DOI: 10.1109/ISCA45697.2020.00078. Cited on page: 110

[131] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *Proceedings of the 47th International Symposium on Computer Architecture*, pages 1050–1063, 2020. DOI: 10.1109/ISCA45697.2020.00089. Cited on page: 103

[132] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing Memory in Heterogeneous Systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 637–650, 2018. DOI: 10.1145/3173162.3173194. Cited on page: 110, 154

[133] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan

Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012. DOI: 10.1109/MM.2011.108. Cited on page: 3, 21, 27, 28

[134] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 444–455, 2016. DOI: 10.1109/ISCA.2016.46. Cited on page: 28

[135] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan, Learning Memory Access Patterns. Cited on page: 112

[136] John L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, 2006. DOI: 10.1145/1186736.1186737. Cited on page: xxix, 63, 67, 88, 163, 167, 168, 182, 183

[137] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. HUB: Hugepage Ballooning in Kernel-Based Virtual Machines. In *Proceedings of the 2018 International Symposium on Memory Systems*, MEMSYS '18, page 31–37, 2018. DOI: 10.1145/3240302.3240420. Cited on page: 31

[138] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. HUB: Hugepage Ballooning in Kernel-Based Virtual Machines. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, pages 31–37, 2018. DOI: 10.1145/3240302.3240420. Cited on page: 140

[139] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, page 465–478, 2016. Cited on page: 31

[140] Jerry Huck and Jim Hays. Architectural Support for Translation Table Management in Large Address Space Machines. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, pages 39–50, 1993. DOI: 10.1145/165123.165128. Cited on page: 38, 85, 110, 138, 154

[141] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc Efficiency to Fleet Efficiency: a Hugepage-aware Memory Allocator. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '21, pages 257–273, 2021. Cited on page: 140

[142] Intel Corporation, TLBs, Paging-Structure Caches, and Their Invalidation, `https://composter.com.ua/documents/TLBs_Paging-Structure_Caches_and_Their_Invalidation.pdf`. Cited on page: 5, 6, 7, 31, 48, 161

[143] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Access Map Pattern Matching for Data Cache Prefetch. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 499–500, 2009. DOI: 10.1145/1542275.1542349. Cited on page: 2, 3, 14, 26, 27, 160

[144] Yasuo Ishii, Mary Inaba, and Kei Hiraki. Unified Memory Optimizing Architecture: Memory Subsystem Control with a Unified Predictor. In *Proceedings of the 26th International Conference on Supercomputing*, ICS '12, page 267–278, 2012. DOI: 10.1145/2304576.2304614. Cited on page: 2, 3, 14, 26, 27

[145] B. Jacob and T. Mudge. Software-Managed Address Translation. In *Proceedings 3rd International Symposium on High-Performance Computer Architecture*, HPCA '97, pages 156–167, 1997. DOI: 10.1109/HPCA.1997.569652. Cited on page: 47

[146] Bruce Jacob and Trevor Mudge. Virtual Memory: Issues of Implementation. *Computer*, 31(6):33–43, 1998. DOI: 10.1109/2.683005. Cited on page: 38

[147] Bruce Jacob and Trevor Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 18(4):60–75, 1998. DOI: 10.1109/40.710872. Cited on page: 38

[148] Bruce L. Jacob. *The Memory System: You Can't Avoid It, You Can't Ignore It, You Can't Fake It*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009. Cited on page: 1, 19

[149] Bruce L. Jacob and Trevor N. Mudge. A Look at Several Memory Management Units, TLB-Refill Mechanisms, and Page Table Organizations. In *Proceedings of the 8th In-*

*ternational Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’98, page 295–306, 1998. DOI: 10.1145/291069.291065. Cited on page: 6, 47

[150] Akanksha Jain and Calvin Lin. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *Proceedings of the 46th International Symposium on Microarchitecture*, MICRO ’13, page 247–259, 2013. DOI: 10.1145/2540708.2540730. Cited on page: 2, 3, 14, 26, 27, 28, 160

[151] A. Jaleel and B. Jacob. In-line Interrupt Handling for Software-managed TLBs. In *Proceedings of the 19th International Conference on Computer Design: VLSI in Computers and Processors*, ICCD ’01, pages 62–67, 2001. DOI: 10.1109/ICCD.2001.955004. Cited on page: 6, 47

[152] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr., and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 43rd International Symposium on Microarchitecture*, MICRO ’10, pages 151–162, 2010. DOI: 10.1109/MICRO.2010.52. Cited on page: 43

[153] James Wang, TLB Prefetching, `https://patents.google.com/patent/US20110010521`. Cited on page: 11, 109, 150

[154] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proceedings of the 23rd SIGSAC Conference on Computer and Communications Security*, CCS ’16, page 380–392, 2016. DOI: 10.1145/2976749.2978321. Cited on page: 49, 161

[155] Gandhi Jayneel. *Efficient Memory Virtualization*. PhD thesis, University of Wisconsin at Madison, 2016. Cited on page: 31

[156] D.A. Jiménez and C. Lin. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA ’01, pages 197–206, 2001. DOI: 10.1109/HPCA.2001.903263. Cited on page: 211

[157] Daniel A. Jiménez and Elvira Teran. Multiperspective Reuse Prediction. In *Proceedings of the 50th International Symposium on Microarchitecture*, MICRO '17, page 436–448, 2017. DOI: 10.1145/3123939.3123942. Cited on page: 184, 211

[158] Doug Joseph and Dirk Grunwald. Prefetching Using Markov Predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, ISCA '97, pages 252–263, 1997. DOI: 10.1145/264107.264207. Cited on page: 105

[159] N.P. Jouppi. Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, ISCA '90, pages 364–373, 1990. DOI: 10.1109/ISCA.1990.134547. Cited on page: 2, 3, 26

[160] T. Juan, T. Lang, and J.J. Navarro. Reducing TLB Power Requirements. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 196–201, 1997. DOI: 10.1145/263272.263332. Cited on page: 31, 46

[161] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating Physical Addresses Directly for Saving Instruction TLB Energy. In *Proceedings of the 35th International Symposium on Microarchitecture*, MICRO '02, pages 185–196, 2002. DOI: 10.1109/MICRO.2002.1176249. Cited on page: 31, 46

[162] I. Kadayif, P. Nath, M. Kandemir, and A. Sivasubramaniam. Compiler-Directed Physical Address Generation for Reducing dTLB Power. In *Proceedings of the 2004 International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 161–168, 2004. DOI: 10.1109/ISPASS.2004.1291368. Cited on page: 31, 46

[163] M. Kandemir, I. Kadayif, and G. Chen. Compiler-Directed Code Restructuring for Reducing Data TLB Energy. In *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '04, page 98–103, 2004. DOI: 10.1145/1016720.1016747. Cited on page: 31, 46

[164] Gokul B. Kandiraju and Anand Sivasubramaniam. Characterizing the D-TLB Behavior of SPEC CPU2000 Benchmarks. In *Proceedings of the 2002 SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 129–139, 2002. DOI: 10.1145/511334.511351. Cited on page: 31

[165] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *Proceedings of the 29th International Symposium on Computer Architecture*, ISCA '02, pages 195–206, 2002. Cited on page: 10, 11, 52, 54, 55, 56, 62, 66, 105, 113, 114, 130, 133, 148, 150

[166] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, pages 158–169, 2015. DOI: 10.1145/2749469.2750392. Cited on page: 4, 12, 31, 41, 46, 61, 113, 114, 115, 118, 120, 140, 142, 161

[167] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal. Energy-efficient Address Translation. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, HPCA '16, pages 631–643, 2016. DOI: 10.1109/HPCA.2016.7446100. Cited on page: 41, 46, 61, 110, 113, 161

[168] Vasileios Karakostas. *Improving the Performance and Energy-efficiency of Virtual Memory*. PhD thesis, Universitat Politecnica de Catalunya, 2016. Cited on page: 31

[169] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, pages 66–78, 2015. DOI: 10.1145/2749469.2749471. Cited on page: 10, 41, 46, 61, 110, 114, 161

[170] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-Cache Coherence. In *Proceedings of the 40th International Symposium on Computer Architecture*, ISCA '13, page 535–546, 2013. DOI: 10.1145/2485922.2485968. Cited on page: 46

[171] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Computer Security Applications Con-*

*ference*, ACSAC '06, pages 339–348, 2006. DOI: 10.1109/ACSAC.2006.9. Cited on page: 31, 35, 49, 161

[172] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti. Path confidence based lookahead prefetching. In *Proceedings of the 49th International Symposium on Microarchitecture*, MICRO '16, pages 1–12, 2016. DOI: 10.1109/MICRO.2016.7783763. Cited on page: xxix, xxx, 2, 3, 14, 16, 26, 27, 29, 107, 139, 160, 162, 164, 169, 170, 181, 184, 186

[173] Donald E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, third edition, 1997. Cited on page: 35, 50, 161

[174] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. LATR: Lazy Translation Coherence. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 651–664, 2018. DOI: 10.1145/3173162.3173198. Cited on page: 31

[175] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting Through the Front-End Bottleneck with Shotgun. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 30–42, 2018. DOI: 10.1145/3173162.3173178. Cited on page: 4, 10, 31, 41, 46, 61, 161

[176] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches Using Spatial Footprints. In *Proceedings of the 25th International Symposium on Computer Architecture*, ISCA '98, pages 357–368, 1998. DOI: 10.1109/ISCA.1998.694794. Cited on page: 14, 27, 28, 160

[177] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 705–721, 2016. Cited on page: 31, 114, 140, 155

[178] H.-H.S. Lee and C.S. Ballapuram. Energy Efficient D-TLB and Data Cache Using Semantic-Aware Multilateral Partitioning. In *Proceedings of the 2003 International*

*Symposium on Low Power Electronics and Design*, ISLPED '05, pages 306–311, 2003. DOI: 10.1109/LPE.2003.1231884. Cited on page: 31

[179] Levinthal D., Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. Intel Performance Analysis Guide, https://www.intel.com/content/dam/develop/external/us/en/documents/performance-analysis-guide-181827.pdf. Cited on page: 3, 21, 27, 28

[180] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B. Brockman, and Norman P. Jouppi. CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '11, pages 694–701, 2011. Cited on page: 86

[181] Yong Li, Rami Melhem, and Alex K. Jones. PS-TLB: Leveraging Page Classification Information for Fast, Scalable and Efficient Translation for Future CMPs. *ACM Trans. Archit. Code Optim.*, 9(4):28:1–28:21, 2013. DOI: 10.1145/2400682.2400687. Cited on page: 137, 177

[182] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine Learning-based Prefetch Optimization for Data Center Applications. In *Proceedings of the 2009 International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '09, pages 1–10, 2009. DOI: 10.1145/1654059.1654116. Cited on page: 112

[183] L. Liu. Multiple-page Translation for TLB. In *Proceedings of 1993 International Conference on Computer Design*, ICCD '93, pages 344–349, 1993. DOI: 10.1109/ICCD.1993.393355. Cited on page: 61, 109

[184] William Lonergan and Paul King. Design of the B 5000 System. *Annals of the History of Computing*, 9(1):16–22, 1987. DOI: 10.1109/MAHC.1987.10000. Cited on page: 31

[185] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. *ACM Trans. Archit. Code Optim.*, 10(1), 2013. DOI: 10.1145/2445572.2445574. Cited on page: 31

[186] Yashwant Marathe, Nagendra Gulur, Jee Ho Ryoo, Shuang Song, and Lizy K. John. CSALT: Context Switch Aware Large TLB. In *Proceedings of the 50th International Symposium on Microarchitecture*, MICRO '17, pages 449–462, 2017. Cited on page: 31

[187] Artemiy Margaritov. *Improving Address Translation Performance in Virtualized Multi-Tenant Systems*. PhD thesis, University of Edinburgh, 2021. Cited on page: 31

[188] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *Proceedings of the 52nd International Symposium on Microarchitecture*, MICRO '19, pages 1023–1036, 2019. DOI: 10.1145/3352460.3358294. Cited on page: 62, 85, 103, 107, 113, 151, 165

[189] Artemiy Margaritov, Dmitrii Ustiugov, Amna Shahab, and Boris Grot. PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 211–223, 2021. DOI: 10.1145/3445814.3446704. Cited on page: 31

[190] Matthias Waldhauer, New AMD Zen core details emerged, `http://dresdenboy.blogspot.com/2016/02/new-amd-zen-core-details-emerged.html`, Accessed: 05-9-2022. Cited on page: 6, 47

[191] Collin McCurdy, Alan L. Cox, and Jeffrey Vetter. Investigating the TLB Behavior of High-end Scientific Applications on Commodity Microprocessors. In *Proceedings of the 2008 International Symposium on Performance Analysis of Systems and Software*, ISPASS '08, pages 95–104, 2008. DOI: 10.1109/ISPASS.2008.4510742. Cited on page: 31

[192] Scott McFarling, Combining Branch Predictors, `https://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf`. Cited on page: 78

[193] Sally A. McKee. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04, page 162, 2004. DOI: 10.1145/977091.977115. Cited on page: 1, 14, 159

[194] Ulrich Meyer and Peter Sanders. Δ-stepping: A Parallelizable Shortest Path Algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. Cited on page: 88

[195] Pierre Michaud. Best-offset Hardware Prefetching. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, HPCA '16, pages 469–480, 2016. DOI: 10.1109/HPCA.2016.7446087. Cited on page: xxix, 2, 3, 14, 16, 26, 27, 107, 160, 162, 164, 184

[196] S. Mirbagher-Ajorpaz, E. Garza, G. Pokam, and D. A. Jiménez. CHiRP: Control-Flow History Reuse Prediction. In *Proceedings of the 2020 53rd International Symposium on Microarchitecture*, MICRO '16, pages 131–145, 2020. DOI: 10.1109/MICRO50266.2020.00023. Cited on page: 89, 119, 140, 141, 142, 155, 184

[197] Sparsh Mittal. A Survey of Recent Prefetching Techniques for Processor Caches. *ACM Computing Surveys*, 49(2), 2016. DOI: 10.1145/2907071. Cited on page: 2, 14, 160

[198] Sparsh Mittal. A Survey of Techniques for Dynamic Branch Prediction. *Concurrency and Computation Practice and Experience*, 31, 2018. DOI: 10.1002/cpe.4666. Cited on page: 2, 84

[199] Alessandro Morari, Roberto Gioiosa, Robert W. Wisniewski, Bryan S. Rosenburg, Todd A. Inglett, and Mateo Valero. Evaluating the Impact of TLB Misses on Future HPC Systems. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 1010–1021, 2012. DOI: 10.1109/IPDPS.2012.94. Cited on page: 31

[200] N. P. Nagendra, G. Ayers, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. *IEEE Micro*, 40(3):56–63, 2020. Cited on page: 114, 115, 118, 120, 142

[201] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, page 27–38, 1993. DOI: 10.1145/165123.165127. Cited on page: 47

[202] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design Tradeoffs for Software-Managed TLBs. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, pages 27–38, 1993. DOI: 10.1145/165123.165127. Cited on page: 4, 10, 31, 61

[203] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '15, pages 1–12, 2015. DOI: 10.1145/2807591.2807626. Cited on page: 2

[204] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 89–104, 2002. Cited on page: 31, 35, 110, 161

[205] Agustín Navarro-Torres, Biswabandan Panda, Jesús Alastruey-Benedé, Pablo Ibáñez, Víctor Viñals-Yúfera, and Alberto Ros. Berti: an accurate local-delta data prefetcher. In *Proceedings of the 55th International Symposium on Microarchitecture*, MICRO '22, pages 975–991, 2022. DOI: 10.1109/MICRO56248.2022.00072. Cited on page: 184

[206] K.J. Nesbit and J.E. Smith. Data Cache Prefetching Using a Global History Buffer. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 96–96, 2004. DOI: 10.1109/HPCA.2004.10030. Cited on page: 2, 3, 26

[207] G. Ottoni and B. Maher. Optimizing Function Placement for Large-scale Data-center Applications. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 233–244, 2017. DOI: 10.1109/CGO.2017.7863743. Cited on page: 13, 114, 141, 155

[208] Samuel Pakalapati and Biswabandan Panda. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *Proceedings of the 2020 47th International Symposium on Computer Architecture*, ISCA '20, pages

118–131, 2020. DOI: 10.1109/ISCA45697.2020.00021. Cited on page: 59, 184, 196, 200

[209] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 International Symposium on Code Generation and Optimization*, CGO '19, pages 2–14, 2019. Cited on page: 12, 113, 114, 115, 118

[210] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making Huge Pages Actually Useful. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 679–692, 2018. DOI: 10.1145/3173162.3173203. Cited on page: 31

[211] Ashish Panwar, Sorav Bansal, and K. Gopinath. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 347–360, 2019. DOI: 10.1145/3297858.3304064. Cited on page: 31

[212] M. Papadopoulou, X. Tong, A. Seznec, and A. Moshovos. Prediction-based Superpage-friendly TLB Designs. In *Proceedings of the 21st International Symposium on High Performance Computer Architecture*, HPCA '15, pages 210–222, 2015. DOI: 10.1109/HPCA.2015.7056034. Cited on page: 10, 31, 84, 110

[213] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage under Diverse Fragmented Memory Allocations. *SIGARCH Comput. Archit. News*, 2017. DOI: 10.1145/3140659.3080217. Cited on page: 31, 41, 46, 61, 110

[214] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *Proceedings of the 2017 44th International Symposium on Computer Architecture*, ISCA '17, pages 444–456, 2017. DOI: 10.1145/3079856.3080217. Cited on page: 110, 113

[215] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990. Cited on page: 1, 2, 7, 18, 19, 44, 48, 84, 137, 161, 209

[216] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic Locality and Context-Based Prefetching Using Reinforcement Learning. In *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, page 285–297, 2015. DOI: 10.1145/2749469.2749473. Cited on page: 112

[217] Leeor Peled, Uri Weiser, and Yoav Etsion. A Neural Network Prefetcher for Arbitrary Memory Access Patterns. *ACM Transactions on Architecture and Code Optimization*, 16(4), 2019. DOI: 10.1145/3345000. Cited on page: 2, 3, 26

[218] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for Accurate and Efficient Simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, 2003. DOI: 10.1145/885651.781076. Cited on page: 89, 184

[219] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, pages 558–567, 2014. DOI: 10.1109/HPCA.2014.6835964. Cited on page: 6, 10, 13, 61, 62, 106, 109, 110, 113, 114, 154

[220] Binh Pham. *Architectural Support for Efficient Virtual Memory on Big-Memory Systems*. PhD thesis, Rutgers University, 2016. Cited on page: 31

[221] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *Proceedings of the 45th International Symposium on Microarchitecture*, MICRO '12, pages 258–269, 2012. DOI: 10.1109/MICRO.2012.32. Cited on page: 6, 13, 61, 62, 106, 109, 110, 113, 114, 154

[222] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO '15, pages 1–12, 2015. DOI: 10.1145/2830772.2830773. Cited on page: 6, 31, 49, 110, 111, 140, 154

[223] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs

with Unified Address Spaces. *SIGARCH Comput. Archit. News*, 42(1):743–758, 2014. DOI: 10.1145/2654822.2541942. Cited on page: 203

[224] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. PLDI 2019, page 31–47, 2019. DOI: 10.1145/3314221.3314637. Cited on page: xxvii, 118

[225] Seth H Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture*, HPCA '14, pages 626–637, 2014. DOI: 10.1109/HPCA.2014.6835971. Cited on page: 2, 3, 26

[226] Xiaogang Qiu and M. Dubois. Towards Virtually-addressed Memory Hierarchies. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 51–62, 2001. DOI: 10.1109/HPCA.2001.903251. Cited on page: 46

[227] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th International Symposium on Microarchitecture*, MICRO '06, pages 423–432, 2006. DOI: 10.1109/MICRO.2006.49. Cited on page: 2

[228] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive Insertion Policies for High Performance Caching. *SIGARCH Computer Architecture News*, 35(2):381–391, 2007. DOI: 10.1145/1273440.1250709. Cited on page: 16, 164, 179, 180, 194

[229] Milan Radulović. *Memory Bandwidth and Latency in HPC: System Requirements and Performance Impact*. PhD thesis, Universitat Politecnica de Catalunya, 2019. Cited on page: 18

[230] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan Qasem. Maximizing Hardware Prefetch Effectiveness with Machine Learning. In *Proceedings of the 17th International Conference on High Performance Computing and Communications, 7th International Symposium on Cyberspace Safety and Security, and 12th International Conference on Embedded Software and Systems*, HPCC-CSS-ICESS '15, pages 383–389, 2015. DOI: 10.1109/HPCC-CSS-ICESS.2015.175. Cited on page: 112

[231] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *Proceedings of the 54th International Symposium on Microarchitecture*, MICRO '21, page 1106–1120, 2021. DOI: 10.1145/3466752.3480062. Cited on page: 41, 46, 161, 166, 168, 200

[232] A. Ramirez, O.J. Santana, J.L. Larriba-Pey, and M. Valero. Fetching Instruction Streams. In *Proceedings of the 35th International Symposium on Microarchitecture*, MICRO '02, pages 371–382, 2002. DOI: 10.1109/MICRO.2002.1176264. Cited on page: 2

[233] G. Reinman, B. Calder, and T. Austin. Fetch Directed Instruction Prefetching. In *Proceedings of the 32nd International Symposium on Microarchitecture*, MICRO '99, pages 16–27, 1999. DOI: 10.1109/MICRO.1999.809439. Cited on page: 115, 155, 156

[234] T.H. Romer, W.H. Ohlrich, A.R. Karlin, and B.N. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *Proceedings of the 22nd International Symposium on Computer Architecture*, ISCA '95, pages 176–187, 1995. DOI: 10.1145/223982.224419. Cited on page: 31

[235] Alberto Ros and Alexandra Jimborean. A Cost-Effective Entangling Prefetcher for Instructions. In *Proceedings of the 48th International Symposium on Computer Architecture*, ISCA '21, pages 99–111, 2021. DOI: 10.1109/ISCA52012.2021.00017. Cited on page: 127

[236] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th Symposium on Operating Systems Principles*, SOSP '95, page 285–298, 1995. DOI: 10.1145/224056.224078. Cited on page: 31

[237] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, 1997. DOI: 10.1145/244804.244807. Cited on page: 4, 5, 10, 31, 41, 46, 61, 161

[238] Amir Roth and Gurindar S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, ISCA '99, page 111–121, 1999. DOI: 10.1145/300979.300989. Cited on page: 29

[239] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '98, page 115–126, 1998. DOI: 10.1145/291069.291034. Cited on page: 29

[240] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence Based Prefetching for Linked Data Structures. *SIGOPS Oper. Syst. Rev.*, 32(5):115–126, 1998. DOI: 10.1145/384265.291034. Cited on page: 29

[241] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *Proceedings of the 44th International Symposium on Computer Architecture*, ISCA '17, pages 469–480, 2017. DOI: 10.1145/3079856.3080210. Cited on page: 61, 109, 113, 154

[242] Dave Sager, Desktop Platforms Group, and Intel Corp. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, 1:2001, 2001. Cited on page: 3, 21, 27, 28

[243] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB Preloading. In *Proceedings of the 27th International Symposium on Computer Architecture*, ISCA '00, pages 117–127, 2000. DOI: 10.1145/339647.339666. Cited on page: 58, 62, 105, 113, 155

[244] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page Overlays:

An Enhanced Virtual Memory Framework to Enable Fine-Grained Memory Management. In *Proceedings of the 42nd International Symposium on Computer Architecture*, ISCA '15, pages 79–91, 2015. DOI: 10.1145/2749469.2750379. Cited on page: 49

[245] A. Seznec, The FNL+MMA Instruction Cache Prefetcher, `https://hal.inria.fr/hal-02884880/document`. Cited on page: 127

[246] A. Seznec. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Transactions on Computers*, 53(7):924–927, 2004. DOI: 10.1109/TC.2004.21. Cited on page: 10, 110

[247] André Seznec. A Case for Two-Way Skewed-Associative Caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, ISCA '93, page 169–178, 1993. DOI: 10.1145/165123.165152. Cited on page: 2

[248] André Seznec. A New Case for the TAGE Branch Predictor. In *Proceedings of the 44th International Symposium on Microarchitecture*, MICRO '11, page 117–127, 2011. DOI: 10.1145/2155620.2155635. Cited on page: 157

[249] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th SIGSAC Conference on Computer and Communications Security*, CCS '04, pages 298–307, 2004. DOI: 10.1145/1030083.1030124. Cited on page: 31, 35, 49, 141, 161

[250] Manish Shah, Robert Golla, Gregory Grohoski, Paul Jordan, Jama Barreh, Jeffrey Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, Zeid Samoail, Matt Smittle, and Thomas Ziaja. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro*, 32(2):8–19, 2012. DOI: 10.1109/MM.2012.1. Cited on page: 35

[251] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO '15, pages 141–152, 2015. DOI: 10.1145/2830772.2830793. Cited on page: xxix, 2, 3, 14, 16, 26, 27, 160, 162, 164, 184

[252] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 861–873, 2021. DOI: 10.1145/3445814.3446752. Cited on page: 211

[253] Yossi Shiloach and Uzi Vishkin, An O (log n) Parallel Connectivity Algorithm, `https://www.sciencedirect.com/science/article/abs/pii/0196677482900086?via%3Dihub`. Cited on page: 88

[254] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. Scheduling Page Table Walks for Irregular GPU Applications. In *Proceedings of the 45th International Symposium on Computer Architecture*, ISCA '18, pages 180–192, 2018. DOI: 10.1109/ISCA.2018.00025. Cited on page: 203

[255] Seunghee Shin, Michael LeBeane, Yan Solihin, and Arkaprava Basu. Neighborhood-Aware Address Translation for Irregular GPU Applications. In *Proceedings of the 51st International Symposium on Microarchitecture*, MICRO-51, pages 352–363, 2018. DOI: 10.1109/MICRO.2018.00036. Cited on page: 62, 109, 203

[256] B. Sinharoy, Ron Kalla, Joel Tendler, R. Eickemeyer, and Jody Joyner. POWER5 System Microarchitecture. *IBM Journal of Research and Development*, 49:505 – 521, 2005. DOI: 10.1147/rd.494.0505. Cited on page: 3, 21, 27, 28

[257] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. PageForge: A Near-Memory Content-Aware Page-Merging Architecture. In *Proceedings of the 50th International Symposium on Microarchitecture*, MICRO '17, pages 302–314, 2017. Cited on page: 31

[258] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. BabelFish: Fusing Address Translations for Containers. In *Proceedings of the 47th International Symposium on Computer Architecture*, ISCA '20, pages 501–514, 2020. DOI: 10.1109/ISCA45697.2020.00049. Cited on page: 31

[259] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pages 1093–1108, 2020. DOI: 10.1145/3373376.3378493. Cited on page: 61, 85, 110, 138, 154

[260] A.J. Smith. Sequential Program Prefetching in Memory Hierarchies. *Computer*, 11 (12):7–21, 1978. DOI: 10.1109/C-M.1978.218016. Cited on page: 2, 3, 26, 27, 54

[261] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46, 2016. DOI: 10.1109/MM.2016.25. Cited on page: 3, 21, 28

[262] S. Somogyi, T.F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. In *Proceedings of the 33rd International Symposium on Computer Architecture*, ISCA '06, pages 252–263, 2006. DOI: 10.1109/ISCA.2006.38. Cited on page: 2, 3, 14, 26, 27, 29, 160

[263] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-Temporal Memory Streaming. *SIGARCH Computer Architecture News*, 37(3):69–80, 2009. DOI: 10.1145/1555815.1555766. Cited on page: 2, 3, 14, 26, 27, 160

[264] Shekhar Srikantaiah and Mahmut Kandemir. Synergistic TLBs for High Performance Address Translation in Chip Multiprocessors. In *2010 43rd International Symposium on Microarchitecture*, pages 313–324, 2010. DOI: 10.1109/MICRO.2010.26. Cited on page: 31

[265] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 63–74, 2007. DOI: 10.1109/HPCA.2007.346185. Cited on page: 2, 3, 26

[266] David Suggs, Mahesh Subramony, and Dan Bouvier. The AMD "Zen 2" Processor. *IEEE Micro*, 40(2):45–52, 2020. DOI: 10.1109/MM.2020.2974217. Cited on page: 3, 21, 28

[267] Mark Swanson, Leigh Stoller, and John Carter. Increasing TLB Reach Using Superpages Backed by Shadow Memory. In *Proceedings of the 25th International Symposium on Computer Architecture*, ISCA '98, pages 204–213, 1998. DOI: 10.1145/279358.279388. Cited on page: 110

[268] M. Talluri, Shing Kong, M.D. Hill, and D.A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings the 19th International Symposium on Computer Architecture*, ISCA '92, pages 415–424, 1992. DOI: 10.1109/ISCA.1992.753337. Cited on page: 35

[269] M. Talluri, M. D. Hill, and Y. A. Khalidi. A New Page Table for 64-Bit Address Spaces. In *Proceedings of the 15th Symposium on Operating Systems Principles*, SOSP '95, page 184–200, 1995. DOI: 10.1145/224056.224071. Cited on page: 35

[270] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '94, pages 171–182, 1994. DOI: 10.1145/195473.195531. Cited on page: 10, 106, 110

[271] Madhusudhan Talluri, Shing Kong, Mark D. Hill, and David A. Patterson. Tradeoffs in Supporting Two Page Sizes. In *Proceedings of the 19th International Symposium on Computer Architecture*, ISCA '92, pages 415–424, 1992. DOI: 10.1145/139669.140406. Cited on page: 10, 110

[272] Dong Tang, P. Carruthers, Z. Totari, and M.W. Shapiro. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. In *Proceedings of the 36th International Conference on Dependable Systems and Networks*, DSN '06, pages 365–370, 2006. DOI: 10.1109/DSN.2006.13. Cited on page: 31

[273] David Tarjan and Kevin Skadron. Merging Path and Gshare Indexing in Perceptron Branch Prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, 2005. DOI: 10.1145/1089008.1089011. Cited on page: 139, 181

[274] George Taylor, Peter Davies, and Michael Farmwald. The TLB Slice—a Low-Cost High-Speed Address Translation Mechanism. In *Proceedings of the 17th Interna-*

*tional Symposium on Computer Architecture*, ISCA '90, page 355–363, 1990. DOI: 10.1145/325164.325161. Cited on page: 45

[275] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002. DOI: 10.1147/rd.461.0005. Cited on page: 27

[276] Stephan van Schaik, Kaveh Razavi, Ben Gras, Herbert Bos, and Cristiano Giuffrida. RevAnC: A Framework for Reverse Engineering Hardware Page Table Caches. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec '17, 2017. DOI: 10.1145/3065913.3065918. Cited on page: 46

[277] Steven P. Vanderwiel and David J. Lilja. Data Prefetch Mechanisms. *ACM Comput. Surv.*, 32(2):174–199, 2000. DOI: 10.1145/358923.358939. Cited on page: 26, 54

[278] Georgios Vavouliotis, Lluc Alvarez, and Marc Casas, Pushing the Envelope on Free TLB Prefetching, https://upcommons.upc.edu/handle/2117/346621. Cited on page: 64

[279] Georgios Vavouliotis, Lluc Alvarez, Daniel Jiménez, and Marc Casas, Cost-Effective Instruction TLB Prefetching, https://gvavou5.github.io/Documents/Vavouliotis_Poster_YArch.pdf. Cited on page: 176

[280] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas, Exploiting Page Table Locality for Agile TLB Prefetching, https://gvavou5.github.io/Documents/ISCA_22_Poster_48x36.pdf. Cited on page: 205

[281] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, and Marc Casas, Leveraging Page Size Information to Enhance Data Cache Prefetching, https://gvavou5.github.io/Documents/Vavouliotis_Poster_ACM21.pdf. Cited on page: 206

[282] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, and Marc Casas, Leveraging Page Size Information to Enhance Data Cache Prefetching, https://gvavou5.github.io/Documents/Vavouliotis_Poster_SRC21.pdf. Cited on page: 206

[283] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. Morrigan: A Composite Instruction TLB Prefetcher. In *Proceedings of the 54th International Symposium on Microarchitecture*, MICRO '21, page 1138–1153, 2021. DOI: 10.1145/3466752.3480049. Cited on page: xv, 176

[284] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. Exploiting Page Table Locality for Agile TLB Prefetching. In *Proceedings of the 48th International Symposium on Computer Architecture*, ISCA '21, pages 85–98, 2021. DOI: 10.1109/ISCA52012.2021.00016. Cited on page: xv, 119, 141, 165

[285] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V. Gratz, Daniel A. Jiménez, and Marc Casas. Page Size Aware Cache Prefetching. In *Proceedings of the 55th International Symposium on Microarchitecture*, MICRO '22, pages 956–974, 2022. DOI: 10.1109/MICRO56248.2022.00070. Cited on page: xv

[286] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*, EuroSys '15, 2015. DOI: 10.1145/2741948.2741964. Cited on page: 142

[287] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *Proceedings of the 2022 Symposium on Security and Privacy*, SP '22, 2022. Cited on page: 14, 29, 59, 161

[288] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 340–349, 2011. DOI: 10.1109/PACT.2011.65. Cited on page: 137, 177

[289] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. BuMP: Bulk Memory Access Prediction and Streaming. In *Proceedings of the 47th International Sympo-*

*sium on Microarchitecture*, MICRO '14, pages 545–557, 2014. DOI: 10.1109/MICRO.2014.44. Cited on page: 14, 27, 160

[290] Haoyuan Wang and Zhiwei Luo, Data Cache Prefetching with Perceptron Learning. Cited on page: 112

[291] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '02, page 304–316, 2002. DOI: 10.1145/605397.605429. Cited on page: 31

[292] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. An In-Cache Address Translation Mechanism. In *Proceedings of the 13th International Symposium on Computer Architecture*, ISCA '86, page 358–365, 1986. Cited on page: 46

[293] Hao Wu, Krishnendra Nathella, Joseph Pusdesris, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Temporal Prefetching Without the Off-Chip Metadata. In *Proceedings of the 52nd International Symposium on Microarchitecture*, MICRO '19, page 996–1008, 2019. DOI: 10.1145/3352460.3358300. Cited on page: 2, 3, 14, 26, 27, 28, 160

[294] Hao Wu, Krishnendra Nathella, Dam Sunwoo, Akanksha Jain, and Calvin Lin. Efficient Metadata Management for Irregular Data Prefetching. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 1–13, 2019. Cited on page: 2, 3, 14, 26, 27, 28, 160

[295] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995. DOI: 10.1145/216585.216588. Cited on page: 1, 14, 18, 159

[296] Jiachen Xue and Mithuna Thottethodi. PreTrans: Reducing TLB CAM-search via Page Number Prediction and Speculative Pre-translation. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 341–346, 2013. DOI: 10.1109/ISLPED.2013.6629320. Cited on page: 110, 154

[297] Z. Yan, J. Vesely, G. Cox, and A. Bhattacharjee. Hardware Translation Coherence for Virtualized Systems. In *Proceedings of the 44th International Symposium on Computer*

*Architecture*, ISCA '17, pages 430–443, 2017. DOI: 10.1145/3079856.3080211. Cited on page: 43

[298] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation Ranger: Operating System Support for Contiguity-Aware TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 698–710, 2019. DOI: 10.1145/3307650.3322223. Cited on page: 10, 31, 41, 46, 61, 89, 140, 161, 168

[299] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. CRAMM: Virtual Memory Support for Garbage-Collected Applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 103–116, 2006. Cited on page: 31, 50

[300] Idan Yaniv and Dan Tsafrir. Hash, Don'T Cache (the Page Table). In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, pages 337–350, 2016. DOI: 10.1145/2896377.2901456. Cited on page: 61, 85, 110, 113, 138, 154

[301] Tse-Yu Yeh and Yale N. Patt. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. *SIGMICRO Newsletter*, 23(1–2):129–139, 1992. DOI: 10.1145/144965.145016. Cited on page: 2

[302] Hongil Yoon and Gurindar S. Sohi. Revisiting Virtual L1 Caches: A Practical Design Using Dynamic Synonym Remapping. In *Proceedings of the 22nd International Symposium on High Performance Computer Architecture*, HPCA '16, pages 212–224, 2016. DOI: 10.1109/HPCA.2016.7446066. Cited on page: 46

[303] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. IMP: Indirect Memory Prefetcher. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO '15, pages 178–190, 2015. DOI: 10.1145/2830772.2830807. Cited on page: 2, 3, 26, 29

[304] Y. Zhou, X. Dong, A. L. Cox, and S. Dwarkadas. On the Impact of Instruction Address Translation Overhead. In *Proceedings of the 2019 International Symposium on Performance Analysis of Systems and Software*, ISPASS '19, pages 106–116, 2019. DOI: 10.1109/ISPASS.2019.00018. Cited on page: 12, 13, 113, 114, 140, 155