IMPROVING INSTRUCTION CACHE PERFORMANCE FOR MODERN PROCESSORS WITH GROWING WORKLOADS

NAYANA PRASAD NAGENDRA

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE
ADVISER: DAVID I. AUGUST

SEPTEMBER 2021

© Copyright by Nayana Prasad Nagendra, 2021.

All Rights Reserved

Abstract

Cloud computing is the backbone of today's digital world. Billions of devices ranging from internet-of-things to mobile phones connect to services running on massive data centers, dubbed as Warehouse Scale Computers (WSCs). Due to the scale at which WSCs operate, their performance has a huge cost impact. Even the slightest percentage improvement in performance generates significant profits for the companies that operate them. Besides, they have a hugely adverse effect on the environment because of their massive carbon footprint.

Surprisingly, our work with Google shows significant inefficiencies among the WSC processor utilization. More than two-thirds of the CPU cycles are wasted in many forms. Specifically, nearly 20% of the wasted cycles come from the processor front-end, which is responsible for keeping the pipeline fed with useful instructions. Since a substantial fraction of the front-end cycle wastages occurs due to instruction cache misses, this Dissertation has taken a full-stack approach to study and solve those.

There have been vast growths in data center workloads and their complexity. Consequently, programs' working set sizes are much larger than those of the instruction caches in modern processors. With the nearing end of Moore's Law, innovations are necessary to increase the efficiency of instruction cache usage, given the growing workloads.

In this Dissertation, we first present AsmDB, a fleet-wide study of Google workloads to understand processor front-end bottlenecks. Based on our knowledge, this is the first work to study the in-depth behavior of instruction caches at such a large scale. Among its many findings, AsmDB highlights inefficient usage of instruction caches because of significant fragmentation even among the highly-optimized code.

The second part of this Dissertation describes Emissary, a new approach to alleviate the said problem by using the available cache capacity more efficiently. Observing that not all cache misses cause stalls, Emissary employs a novel cache replacement algorithm that reserves cache capacity for lines whose misses cause the most significant performance

impact. Emissary outperforms all known cache replacement algorithms in performance and energy consumption. In some cases, it produces speedups that exceed Belady's OPT, the perfect-knowledge minimal miss ideal cache replacement algorithm.

Acknowledgments

"Find a great mentor who believes in you, your life will change forever!"

- Bill Walsh

I am grateful for having found amazing mentors and friends during my Ph.D., who believed in me much more than myself and changed my life for good. As I write this Dissertation, I would like to take a moment to thank each one of them for making my Ph.D. journey something that I would treasure for life!

First and foremost, I would like to thank my doctoral adviser, Professor David August, for his endless support over the last several years. I always find a shortage of words to thank him for the amount of care he provides, more than what our family would. His unwavering belief in me at all times has made me grow as a stronger woman, and I am indebted to him. I am forever grateful for his strong recommendation in many places. He is a role model to look upon in every aspect, whether research, technical, communication, or social. I would also like to thank him for his infinite patience to hear me carefully, even though my toddler's screams surrounded me. I have learned several things from him during my time at Princeton, most importantly, approaching a research problem by breaking it down to its simplest form and thinking holistically. I truly appreciate his time reading, providing detailed feedback, and improvising this Dissertation, papers, job talk, and many other materials. His commitment to always aim for the best has made me a better researcher, and I am truly grateful for having found Professor August, an exemplary person, as my mentor.

I will fail in my duties if I can't thank my mentor, Dr. Tipp Moseley. Tipp has been the backbone of my Ph.D. career, with his immense amount of guidance and support at all times. He is a cheerful mentor whose passionate approach makes the time spent with him a fun-filled and rewarding experience. His belief in me and my work has led to many fruitful instances. One such example is receiving the "Top Picks", a relatively rare appreciation received in the community, for our work on AsmDB, the paper presented in ISCA'19. In

addition, Tipp has been tremendously supportive with everything that I have asked him, from recommending me highly in several places to serving on my thesis committee. My career path wouldn't have been the same if I didn't have a chance to intern at Google under the mentorship of Tipp, and I am forever thankful to him.

I want to thank the other members of my thesis committee, Professor David Wentzlaff and Professor Sharad Malik. I would especially like to thank Professor Wentzlaff for recommending me highly for full-time positions and providing thorough feedback on the research in several instances. Additionally, I am also grateful to Professor Simone Campanoni from Northwestern University for his detailed feedback in both the content and the presentation in many instances. I am eternally thankful to my several teachers who have trained and encouraged me to always aim for the best. I always have been building my strengths over the foundation that they created early on.

I thank the Liberty Research Group members I overlapped with and made great friends more than colleagues. I truly enjoyed spending my time at Princeton with Feng Liu, Taewook Oh, Stephen Beard, Soumyadeep Ghosh, Jordan Fix, Heejin Ahn, Sophie Qiu, Hansen Zhang, Sergiy Popovych, Sotiris Apostolakis, Bhargav Reddy Godala, Ziyang Xu, Greg Chan, Susan Tan, Charles Smith, Ishita Chaturvedi, and more recently Yebin. Our lunch/dinner getaways and get-togethers at our homes every once in a while were great stress busters. I always cherish the random one-on-one conversations with Soumyadeep, Heejin, and Bhargav. I want to thank Bhargav, Charles, Ziyang, and Susan for helping me out with my paper submissions and perfecting me with many fine-grained refinements. I am thankful for being part of such a collaborative group.

I have thoroughly enjoyed the two Summers that I spent at Google as an intern, and I want to thank the several members for assisting me in numerous ways. I especially have immense gratitude and appreciation for Svilen Kanev, who has been a great friend and a fantastic mentor. He has guided me in several ways, starting from explaining the massive profiling framework to writing the paper. Additionally, he has also played an enormous

role in shaping the research ideas for Emissary. I thank my manager, Hyoun Kyu Cho, for assisting me with the AsmDB framework. I am grateful for being mentored by Trivikram Krishnamurthy on the detailed aspects of Branch Prediction, which paved the way for my micro-architectural studies later on. Finally, the paper wouldn't have been complete without the efforts of Grant Ayers and Heiner Litz, and I would like to thank them.

I feel fortunate to be mentored by Dr. Parthasarathy Ranganathan, the finest and proficient person in the architecture community. His words of wisdom and appreciation are always fresh in my mind to motivate me even when I have rough days. Likewise, I want to thank Professor Christos Kozyrakis for being supportive in many different ways and cheering for us in every moment.

I am lucky enough to be funded and mentored by great minds at Intel. Our frequent meetings with Dr. Jared Stark have been a steering wheel for my research work, and I truly appreciate his time and detailed feedback. I particularly want to thank Sreenivas Subramoney, Director of Processor Architecture Research Lab, Intel Labs, for his immense support in various forms throughout my Ph.D. journey. Dr. Niranjan Soundararajan from Intel has also been a great support system, and I would like to thank him.

I appreciate the help and support from many others in the Department of Computer Science at Princeton. First, I want to thank Professor Margaret Martonosi and Professor Aarti Gupta for their kind words and moral support. The email that I received from Professor Margaret regarding my acceptance to Princeton cheers me up in any instance. Next, I genuinely appreciate Professor David Dobkin, Maia Ginsburg, Christopher Moretti, and Christopher Brinton for guiding me when I was a teaching assistant for them. I fondly remember the time spent with Professor Dobkin for his warmth and cheerfulness.

I am thankful to Nicki Mahler, a great friend who has supported various administrative tasks from teaching assignments to student status-related questions. I also want to thank Judith Farquer, Michele Brown, and Pamela DelOrefice for helping me out in multiple instances. In addition, I am thankful to Lidia Stokman and Lori Bailey from Electrical

Engineering for scheduling my pre-FPO and FPO. I appreciate the support I received from Davis International Center, especially Katherine Sferra. I am forever thankful for the support from both CS-Staff and CSES in assisting me with several computing-related issues. Additionally, they enabled me to run thousands of experiments on Princeton University clusters, results of which are the primary core of this Dissertation.

I am incredibly grateful to my Indian friends at Princeton. I cherish the moments I had with Tejal Bhamre, my roommate. Moments spent with Tejal, Jahnavi Punekar, Yogesh Goyal, and Srinivas Narayana Ganapathy (NG) over chai have been very relaxing. NG has been more than a brother to me, and I am forever grateful for his advice on several topics. I especially want to thank Sneha Rath and Shantanu Agarwal, for letting me and my kid stay at their place. Soumyadeep Ghosh, Chinmay Khandekar, Debajit Bhattacharya, Ravi Tandon, and Tanya Gupta have brought much more fun into my life. All of them brought me closer to my roots and made my time at Princeton a remarkable one.

Additionally, I have made several great friends at Princeton. Firstly, Themistoklis Melissaris has been a great support system throughout. I want to thank Themis for sticking with me through thick and thin. I am thankful to Yaqiong Li, Marcela Melara, and Swati Roy for always being by my side.

I am indebted to my best friend, Ujwal Hegde, for motivating me to apply for a Ph.D. program and, more importantly, for being there for me always. Furthermore, I am grateful to Professor Steven Freear, Anindyasundar Nandi, and Archana Srinivas for their recommendation to support my Ph.D. applications. Finally, I appreciate the assistance of Merwyn Paul in applying for graduate studies. I am writing this Dissertation today only because of the support from all of them many years ago.

Staying away from family, I found a new family with Mala Ramachandra, Hemant Ramachandra, and their children. I am forever grateful to them for taking care of me as their daughter and always making me feel at home.

I am fortunate to have amazing parents-in-law Lakshmi and Viyyanna, who always treat

me as a daughter and consistently encourage me to achieve greater heights. Likewise, I am thankful to my sister-in-law Sharada for her love and care more than a sister.

As there is a saying, parents are our first teachers. I am blessed to have amazing parents Dr. Nagendra Prasad and Manjula. They have been incredibly supportive throughout my life, often at the cost of their sacrifices. From a young age, they have taught me the importance of education and that the sky is the limit to achieve. I draw great inspiration from them, especially my father, who is my role model. Additionally, I am also thankful to my late grandfather, Dr. Basavaraju, for his immense support and encouragement. I am sure he is proud of me, from wherever he is.

I am fortunate to have the best daughters one could ask for, Isha and Darsha. I would especially like to thank Isha for understanding and supporting me in many situations when I couldn't be my complete self to her. I am grateful to have them both in my life, for they fill my heart with love and joy in every moment and continuously keep me on my toes!

Saving the best for the end, I would like to express my sincere gratitude to my best friend for life and husband, Dr. Vivekanand Kalaparthi, to whom I am eternally indebted. Vivek is my greatest strength, and I am the luckiest to have him in my life. Having experienced the path of a Ph.D. himself, he exceptionally mentors me constantly to be the best version of myself. Vivek, thank you from the bottom of my heart for your immense patience, unconditional love, and for bringing so much joy into our lives every day. I am excited to grow old with you and experience our future unfold.

Thank you to Vivek, Isha, Darsha, Mummy, and Pappa for all of your unconditional love, support, and encouragement to pursue it to the highest level! I could not have done this without your assistance. I dedicate this Ph.D. to all of you.

I gratefully acknowledge the generous financial support for my graduate school work by a variety of sources: the Google Faculty Research Award; "A Frontend for all Workloads", Intel University Research Program, Faculty Grant; and "The Whole Program Critical Path Approach to Parallelism", (NSF Award CCF-1814654).

To my caring parents, *Manjula* and *Dr. Nagendra Prasad*.

To my eternal grand-father, *Dr. Basavaraju*.

To my loving husband, *Dr. Vivekanand*.

And to my beautiful daughters, *Isha* and *Darsha*.

Pillars of my life!

Contents

Abstract . . . iii

Acknowledgments . . . v

List of Tables . . . xiv

List of Figures . . . xv

1 Introduction . . . 1

1.1 Enormous Reliance on Cloud Computing . . . 1

1.2 Modern Hardware is Unable to Meet the Needs of Growing Software . . . 2

1.3 Instruction Caches will Never Have Enough Capacity . . . 4

1.4 Dissertation Contributions . . . 4

1.4.1 Instruction Cache Miss Characterization in a WSC . . . 5

1.4.2 Variance in the Cost of Instruction Cache Misses . . . 5

1.4.3 Cost-Aware Instruction Cache Replacement Policy . . . 6

1.5 Published Material . . . 7

1.6 Dissertation Organization . . . 8

2 Instruction Cache Misses in a WSC . . . 9

2.1 `AsmDB`, an Assembly Database . . . 10

2.2 Where Are The Misses Coming From? . . . 12

2.2.1 Miss Working Sets . . . 12

2.2.2 Miss-Causing Instructions 13
2.3 Code Bloat and Fragmentation 17
2.3.1 Code Bloat 18
2.3.2 Intra-Function Fragmentation 19
2.3.3 Intra-Cacheline Fragmentation 20
2.3.4 Memcmp and the Perils of Micro-Optimization 22
3 Instruction Cache Miss Cost in Modern Processors 26
3.1 Replacement Policies Dictate Cache Performance 27
3.2 Prior Cost-Aware Replacement Policies for Data Cache 29
3.3 Why is Instruction Cache Miss Cost Important? 30
3.4 Decode Starvation as a Cost Metric 31
4 Cost-Aware Instruction Cache Replacement Policy 34
4.1 An Overview Tour of EMISSARY 36
4.2 The EMISSARY Policies 39
4.2.1 Mode Selection 40
4.2.2 Mode Treatment 40
4.2.3 Cache Replacement Policies 42
4.3 Experimental Exploration 42
4.3.1 Machine Model and Simulator 43
4.3.2 Benchmarks 47
4.3.3 Policy Selection and Parameterization 48
4.3.4 Performance 52
4.3.5 Energy Savings 76
4.4 Mechanisms to Improve Performance When Caches Are Big 78
4.5 A Closer Look 80
4.5.1 Speedup Without Starvation Reduction 80

4.5.2 Persistence, By Itself, Improves Hit Rate 84
4.6 Related Work 85
4.6.1 Cost-Aware Cache Replacement Policies 85
4.6.2 Policies Challenging LRU 86
4.6.3 OPT Inspired Strategies 88
5 Concluding Remarks and Future Directions 90
5.1 Future work 91

List of Tables

2.1 Instruction miss intensities for web search 15

4.1 Mode Selection Options 40

4.2 Mode Treatment Options 41

4.3 Realizable cache replacement policies 43

4.4 Processor configurations 44

4.5 Geomean speedup across all configurations for various values of r and N when run on a system with 8kB L1 I-cache. 49

4.6 Geomean speedup across all configurations for various values of r and N when run on a system with 16kB L1 I-cache. 50

4.7 Geomean speedup across all configurations for various values of r and N when run on a system with 32kB L1 I-cache. 51

4.8 Contexts in which P(6):S&R(1/32) beats OPT. 75

List of Figures

1.1 The multicore sequential performance gap 3

2.1 CPU performance potential breakdown (Top-Down) on a web search binary. 9

2.2 Example AsmDB SQL query which ranks x86 extensions (e.g. SSE, AVX) by execution frequency across our WSC fleet. 11

2.3 Fleet-wide distribution of executed instructions, L1-, and L2-instruction misses over unique cache lines. Like instructions, misses also follow a long tail. 13

2.4 Control-flow instruction mixes for several WSC workloads. The remaining 80+% are sequential instructions. 13

2.5 Instructions that lead to i-cache misses on a web search binary. 14

2.6 Fleetwide distribution of jump target distances. 16

2.7 Cumulative distribution of number of targets for indirect jumps and calls. . 17

2.8 Normalized execution frequency vs. function size for the top 100 hottest fleetwide functions. memcmp is a clear outlier. 18

2.9 Distribution of execution over function size. 19

2.10 Maximum inlining depth versus function size for the 100 hottest fleetwide functions. 20

2.11 Fraction of hot code within a function among the 100 hottest fleetwide functions. From left to right, "hot code" defined as covering 90%, 99% and 99.9% of execution. 21
2.12 Intra-cacheline fragmentation vs function size for hotness thresholds of 90%, and 99%. 22
2.13 Instruction execution profile for `memcmp`. 90% of dynamic instructions are contained in 2 cache lines; covering 99% of instructions requires 41 L1I cache lines. 23

3.1 L1I cache accesses in 445.gobmk benchmark on hardware scaled to match WSC behavior. 28
3.2 Percentage of L1I misses causing decode starvation (first bar) and the distribution of L1I misses according to the decode starvation rate of their addresses (second bar), when run on a system with an 8kB L1I. 32

4.1 Performance, misses, and commit-path decode starvations of various cache replacement policies for 400.perlbench on a 32kB 8-way L1I cache. 37
4.2 Frontend microarchitecture 45
4.3 Pipeline diagram depicting out-of-order memory responses received. 47
4.4 Speedup vs. MPKI and Speedup vs. Change in Starvation for 126.gcc . . . 55
4.5 Speedup vs. MPKI and Speedup vs. Change in Starvation for 132.ijpeg . . 56
4.6 Speedup vs. MPKI and Speedup vs. Change in Starvation for 171.swim . . 57
4.7 Speedup vs. MPKI and Speedup vs. Change in Starvation for 175.vpr . . . 58
4.8 Speedup vs. MPKI and Speedup vs. Change in Starvation for 176.gcc . . . 59
4.9 Speedup vs. MPKI and Speedup vs. Change in Starvation for 197.parser . . 60
4.10 Speedup vs. MPKI and Speedup vs. Change in Starvation for 253.perlbmk . 61
4.11 Speedup vs. MPKI and Speedup vs. Change in Starvation for 254.gap . . . 62
4.12 Speedup vs. MPKI and Speedup vs. Change in Starvation for 300.twolf . . 63

4.13 Speedup vs. MPKI and Speedup vs. Change in Starvation for 400.perlbench 64
4.14 Speedup vs. MPKI and Speedup vs. Change in Starvation for 401.bzip2 . . 65
4.15 Speedup vs. MPKI and Speedup vs. Change in Starvation for 444.namd . . 66
4.16 Speedup vs. MPKI and Speedup vs. Change in Starvation for 445.gobmk . 67
4.17 Speedup vs. MPKI and Speedup vs. Change in Starvation for 456.hmmer . 68
4.18 Speedup vs. MPKI and Speedup vs. Change in Starvation for 458.sjeng . . 69
4.19 Speedup vs. MPKI and Speedup vs. Change in Starvation for 464.h264ref . 70
4.20 Speedup vs. MPKI and Speedup vs. Change in Starvation for 541.leela . . . 71
4.21 Speedup vs. MPKI and Speedup vs. Change in Starvation for Xapian (Tailbench) 72
4.22 Speedup and Energy Reduction of a range of techniques relative to LRU. The programs are generally sorted by their baseline MPKI. 73
4.23 Speedup and Energy Reduction of a range of techniques relative to LRU, when simulated with the 'Default Gem5' Model for the first 100 million instructions of each program 74
4.24 Speedup of EMISSARY (P(6):S&R(1/32)) relative to LRU depicted for all the programs when run on architecture with 32kB L1I cache, 8k BTB entries, and 20 FTQ entries. 75
4.25 Speedup of EMISSARY (P(6):S&R(1/32)) relative to LRU depicted at each checkpoint, collected with a fixed interval. Both the programs were run to completion on a system with an 8kB L1I cache. The average IPC is 1.7 and 3.6 for 458.sjeng and 541.leela, respectively, depicted as a dotted line in the respective color. 77
4.26 Set Dynamic Emissary with a single training phase of 50K accesses per set . 81
4.27 Set Dynamic Emissary with a training phase of 50K accesses per set repeating after 500K accesses 81

4.28 Performance, commit-path decode starvations, and the distribution of starvation cycles observed by each program address on the commit path of LRU and EMISSARY policy (P(7):S&H(8)) for 400.perlbench on a 32kB, 8-way L1I cache. 82

Chapter 1

Introduction

Over the last decade, technology scaling has paved the way for the diverging trend in the computing paradigm that has never been experienced before. Smaller, power-efficient mobile devices and internet-of-things (IOT) form one side of the spectrum, where the massive “cloud computing” supports them on the other side. Cloud computing is the true backbone of today’s internet services and has transformed the world into a “Global Village”. Most importantly, it has been serving the needs of all, whether for email, online maps, social networks, e-commerce, video-streaming, or web-search. Such a transformation has also enabled self-driving car technology, space technology, medicine, and drug discovery. More than 4.7 billion people, which is 60% of the global population, are now connected to the internet [15] due to the growing popularity of cloud computing. Moreover, the market is expected to grow to half a trillion dollars in the next few years [7].

1.1 Enormous Reliance on Cloud Computing

Owing to the tremendous growth in the reliance on cloud computing, Google, for example, processes around 40,000 search queries per second, which is more than 3.5 billion searches every day and 1.2 trillion searches every year [16]. Several reports describe a similar trend for active users in social networking and video-streaming platforms [10]. In addition to the

users, IOT devices have grown to tens of billions in number [20, 31] and connect to services running on massive data centers, also called Warehouse Scale Computers (WSCs).

Moreover, every single device connected to WSC demands to receive the response instantaneously. Several reports highlight that many users abandon video playback by waiting just three seconds. Even a hundred milli-seconds latency leads to millions of dollars lost in sales for companies like Amazon [2]. Not just the cost, the performance of WSC has a considerable impact on our planet due to their massive carbon footprint, which is even more significant than the airline industry [17, 21].

As mentioned by Barroso [7, 22], "The data center is the computer," which comprises several thousand computing nodes and whose operating costs are in the order of several hundreds of millions of dollars. Because of the scale at which they operate, even a single percentage improvement in the performance of WSCs leads to several millions of dollars in savings in cost and energy [6, 46].

1.2 Modern Hardware is Unable to Meet the Needs of Growing Software

As a result of the increasing popularity of cloud computing, the complexity of the applications running on WSCs and the size of the datasets they process have been ever-growing. These complex applications rely upon higher-performing and energy-efficient processors to keep up with the latency demands. Semiconductor miniaturization has been at the forefront in realizing these objectives. However, as shown in figure 1.1, Moore's Law is finally approaching its limits [23, 41]. Modern processors comprise billions of transistors, and additional transistors have produced diminishing returns. Single thread performance improvement has slowed to an extent where we are currently lagging behind by 10 years or 41 times from the projected growth. Consequently, it has become highly challenging for the hardware to keep up with the growing demands from the software.

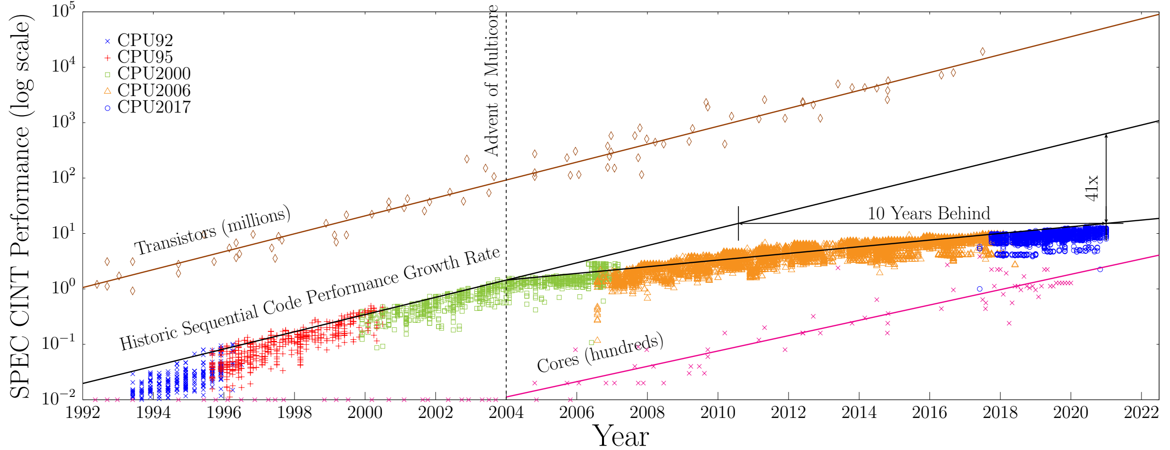


Figure 1.1: The multicore sequential performance gap

General-purpose processors started as simple, in-order, latency-intolerant, and single core. Today, they are complex, out-of-order, latency-tolerant, and many-core. Modern processors deliver performance by avoiding stalls with prediction and by hiding stalls with other valuable tasks. Conventional work in the field has focused on doing more of the same, such as improving branch predictors and cache replacement algorithms with diminishing returns. Despite decades of improvements, our work with Google shows that retiring instructions (i.e., useful work) in Google's highly optimized web-search application account for only 32% of the CPU cycles. In comparison, the remaining 68% of the cycles are wasted in one way or another [6, 46].

Surprisingly, the front-end of the processor, responsible for instruction fetch and keeping the processor pipeline fed accounts for nearly 20% of the wasted cycles. Most of the wastage in the front-end occurs majorly due to instruction cache misses, instruction Translation Lookaside Buffer (TLB) misses, and branch re-steers. Ultimately, the front-end performance limits the maximum achievable efficiency of the entire processor pipeline since no work can be performed if instructions are not fed to the different CPU stages [22, 59].

1.3 Instruction Caches will Never Have Enough Capacity

Multi-level caches are crucial for modern processors that aim to bridge the gap between the fast CPU and the slow memory. However, caches are already resource-constrained and will never have enough capacity to accommodate today's fast-growing workloads. Deep software stacks in which individual requests can traverse many layers of data retrieval, data processing, communication, logging, and monitoring characterize WSC workloads. As a result, instruction footprints have been growing for decades, at rates of over 20% per year [32]. Today, they are 100 times larger than an L1 instruction cache (L1I cache) size and can easily overwhelm it [5]. Thus, instruction cache misses rates encountered by WSC workloads are orders of magnitude larger than the traditionally studied SPEC CPU benchmarks [19].

Because the performance of a general-purpose processor is critically dependent on its ability to feed itself with valuable instructions, poor L1I cache performance manifests itself in significant unrealized performance gains due to front-end stalls. Although it is reasonable to assume that a significant fraction of L1I cache misses gets served from L3 caches in the present-day context, latency from L3 is significant enough to introduce stalls in the processor pipeline.

1.4 Dissertation Contributions

This Dissertation has taken a full-stack approach to understand where the front-end cycle wastage comes from in a WSC and provides a simple yet efficient solution to reduce some of them.

The first part of this Dissertation presents AsmDB, wherein we have studied the root causes of L1I cache misses at the breadth of a WSC. By doing so, we have provided solutions both at the compiler and the architectural levels to reduce front-end bottlenecks for all workloads. We approach the problem by first understanding the interplay between the extensive,

complex, highly optimized, and fast-growing software system globally and the modern hardware. The deep-down study of several hundreds of millions of instructions reveals specific patterns in how instruction caches are utilized in a WSC. Based on the insights gained from the study of WSC workloads and other workloads that exhibit similar behavior in the front-end, we present a first-of-its-kind cost-aware instruction cache replacement policy called Emissary. It proves to be well-performing in all cases and proportionately energy saving, with just a simple modification to Least Recently Used (LRU) replacement algorithm and a single additional bit to each cache entry. Thus, making it a profitable choice for future processor designs.

We briefly discuss each of the contributions below.

1.4.1 Instruction Cache Miss Characterization in a WSC

The large instruction working sets of WSC workloads lead to frequent instruction cache misses and costs in the millions of dollars. Based on our knowledge, this is the first work to analyze where the misses come from and the opportunities to improve them at the WSC scale. Based on a longitudinal analysis of several hundreds of millions of instructions from real-world online services collected over several years, we present two detailed insights on the sources of front-end stalls. First, cold code brought in along with hot code leads to significant cache fragmentation and a correspondingly large number of instruction cache misses. Cache fragmentation is at both the function-level granularity (groups of cache lines) and the cache line granularity (individual cache line). Second, distant branches and calls that are not amenable to traditional cache locality or next-line prefetching strategies account for a significant fraction of cache misses.

1.4.2 Variance in the Cost of Instruction Cache Misses

For decades, researchers have primarily looked at various cache policies to reduce misses. Some have looked to Belady's OPT algorithm [8] for inspiration because it produces the

minimum number of misses using future knowledge. While reducing misses has been the singular focus of many researchers, this is not the path to achieving the best performance. Contrary to conventional wisdom, not all cache misses have the same performance cost, and some have no impact on cost at all in a modern processor, because of their latency tolerating capabilities. Such is true for both the instruction and the data caches, especially when missed in L1 caches and served from L2 caches.

We study varied SPEC and open-source datacenter benchmarks on a simulator we extended to represent the modern fetch engine best to show that a full quarter of L1 instruction cache misses never cause performance bottlenecks. Certain misses stall the processor pipeline always, another category of misses that stall sometimes, and the last category that never stalls. Additionally, the miss cost is closely associated with the branch prediction and targets. Furthermore, there is strong bimodality among the misses, i.e., those misses that stall the pipeline tend to do so every time they miss, and those that do not stall the pipeline tend not to do so every time. Our detailed study enabled us to reduce misses that have a considerable performance impact rather than reducing the overall L1I misses. Based on our knowledge, this is the first work to study the cost impact of instruction cache misses.

1.4.3 Cost-Aware Instruction Cache Replacement Policy

While others [35, 53, 63] have observed that not all cache misses have the same cost, we have demonstrated the first instruction cache replacement policy to leverage it. This cache replacement policy, called Emissary, uses a simple mechanism to reduce the total cost of misses to outperform existing policies focused on miss reduction reliably.

Emissary relies upon decode starvation, a state in which the decode stage is starved of instructions even though it has enough capacity to decode to identify the instruction cache misses that are expensive. We show that decode starvation is a clear indicator for front-end bottlenecks. Moreover, it is straightforward to observe decode starvation events, as the signal already exists in modern processors. In addition, the processor is also aware of the

address that leads to the decode starvation. Given the bimodality of misses, we show that by simply extending every cache entry to include the priority information based on its miss cost, caches get well utilized.

In some cases, Emissary beats the unrealizable Belady's OPT algorithm in terms of both performance and energy (with more but inexpensive misses - those tolerated by the system by executing other instructions). Since many additional transistors Moore's law has given us has been spent on caches to reduce misses for ever-growing workloads, work like this has the potential to free up large fractions of chip real estate for other purposes.

1.5 Published Material

All the material presented in this Dissertation are based on the research work in prior publications.

The instruction cache miss characterization (Chapter 2) has been published in [6, 46]:

> Grant Ayers*, Nayana P. Nagendra*, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan, *Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers.* In the Proceedings of the 46th International Symposium on Computer Architecture (ISCA), 2019
>
> Nayana P. Nagendra*, Grant Ayers*, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan, *Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers.* In IEEE Micro, June 2020.
>
> IEEE Micro's "Top Picks" special issue "based on novelty and potential for long-term impact in the field of computer architecture" in 2019.
>
> * - Join first authors

The cost-aware instruction cache replacement policy, Emissary (Chapters 3 and 4) is currently in submission.

Nayana P. Nagendra, Bhargav Reddy Godala, Charles M. Smith, Ziyang Xu, Greg Chan, Zujun Tan, Svilen Kanev, Tipp Moseley, David I. August, *EMISSARY: Enhanced Miss Awareness Replacement Policy for I-Cache.* In Submission

1.6 Dissertation Organization

Chapter 2 details the longitudinal study of L1I cache misses across Google fleet, at both instruction-level and function-level granularity. Chapter 3 describes the modern front-end and explains the variation among L1I cache miss cost on the processor performance. Chapter 4 proposes the cost-aware L1I cache replacement policy, describes its implementation, presents the results on several SPEC and open-source data center benchmarks, and compares against the related works published in this space. Chapter 5 concludes the Dissertation.

Chapter 2

Instruction Cache Misses in a WSC

The instruction footprint of WSC workloads in particular is often over 100 times larger than the size of a L1I cache and can easily overwhelm it [5]. Studies show it expanding at rates of over 20% per year [32]. This results in L1I cache miss rates which are orders of magnitude higher than the worst cases in desktop-class applications, commonly represented by SPEC CPU benchmarks [19].

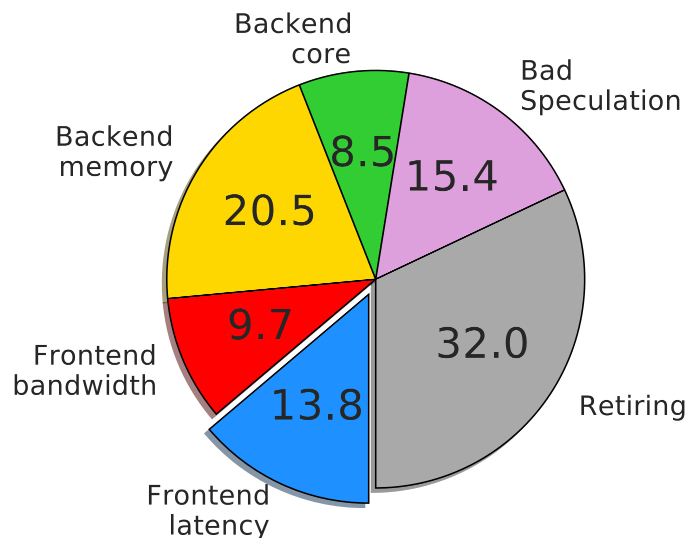


Figure 2.1: CPU performance potential breakdown (Top-Down) on a web search binary.

Because the performance of a general-purpose processor is critically dependent on its ability to feed itself with useful instructions, poor L1I cache performance manifests itself in

large unrealized performance gains due to front-end stalls. In the `AsmDB` paper [6, 46], we corroborate this challenge for Google WSCs on a web search binary. Figure 2.1 presents a Top-Down [68] breakdown of a web search loadtest running on an Intel Haswell CPU. 13.8% of total performance potential is wasted due to "Front-end latency", which is dominated by L1I cache misses. We also measured L1I cache miss rates of 11 misses per kilo-instruction (MPKI). Using the simulation methodology described in the paper [6], we measured a hot steady-state instruction working set of approximately 4 MB. This is significantly larger than L1I caches and L2 caches on today's server CPUs, but small and hot enough to easily fit and remain in the shared L3 cache (typically 10s of MB). Thus, it is reasonable to assume that most L1I cache misses are filled by the L3 cache in the worst case.

In this chapter, we focus on understanding and improving the L1I cache behavior of WSC applications. Specifically, we focus on tools and techniques for "broad" acceleration of thousands of WSC workloads. At the scale of a typical WSC server fleet, performance improvements of few percentage points (and even sub-1% improvements) lead to millions of dollars in cost and energy savings, as long as they are widely applicable across workloads.

2.1 `AsmDB`, an Assembly Database

In order to enable the necessary horizontal analysis and optimization across the server fleet, the Google Wide Profiling (GWP) team built a continuously-updated Assembly Database (`AsmDB`) with instruction- and basic-block-level information for most observed CPU cycles across the thousands of production binaries executing them. Collecting and processing profiling data from hundreds of thousands of machines is a daunting task by itself. In the `AsmDB` paper [6], we present the architecture of a system that can capture and process profiling data in a cost-efficient way, while generating terabytes of data each week.

The `AsmDB` was built with the loftier coverage goals to collect assembly-level information for nearly every instruction executed in Google WSCs in an easy-to-query format.

AsmDB aggregates instruction- and control-flow-data collected from hundreds of thousands of machines each day, and grows by multiple TB each week. GWP has been continuously populating AsmDB over several years.

Schema. AsmDB is a horizontal columnar database. Each row represents a unique instruction along with observed dynamic information from production – the containing basic block's execution counts, as well as any observed prior and next instructions. Each row also contains disassembled metadata for the instruction (assembly opcode, number/type of operands, etc.). This makes population studies trivial, as illustrated by the query in the figure 2.2 which ranks the relative usage of x87/SSE/AVX/etc. instruction set extensions. In addition, each row has metadata for the function and binary containing every instruction, which allows for queries that are more narrowly targeted than the full fleet (§2.3.4). Finally, each instruction is tagged with loop- and basic-block-level information from dynamically-reconstructed control-flow graphs (CFGs). This enables much more complex queries that use full control-flow information (§2.3).

```
SELECT
  SUM(count) /
    (SELECT SUM(count) FROM ASMDB.last3days)
  AS execution_frac,
  asm.feature_name AS feature
FROM ASMDB.last3days
GROUP BY feature ORDER BY execution_frac DESC;
```

Figure 2.2: Example AsmDB SQL query which ranks x86 extensions (e.g. SSE, AVX) by execution frequency across our WSC fleet.

We further correlate AsmDB with hardware performance counter profiles collected by a datacenter-wide profiling system – Google-Wide Profiling (GWP) [57] – in order to reason about specific patterns that affect front-end performance. Since AsmDB contains information pertaining to instructions from across the WSC, our natural first step is to perform detailed study at the granularity of instructions, which are described in §2.2. However, as there are no noticeable pattern evident at the instruction-level granularity, we studied the L1I cache

behavior among groups of instructions. Functions form the natural grouping of instructions, and enabled our studies at the coarser level which are described in §2.3.

2.2 Where Are The Misses Coming From?

We begin our investigation into front-end stalls by characterizing and root-causing L1I cache misses. We first use fleetwide miss profiles to confirm that, as many other WSC phenomena, L1I cache misses also follow a long tail, and sooner or later that must be addressed by some form of automation. We then start looking for patterns that automated optimization can exploit by combining datasets from Google-Wide Profiling (GWP) and `AsmDB`. We focus on *miss-causing* instructions and find that indirect control-flow, as well as distant calls and branches are much more likely to be the root causes for misses.

2.2.1 Miss Working Sets

Working set sizes can tell us in broad strokes how to prioritize optimization efforts. For example, image processing workloads typically have tiny instruction working sets and manually hand-optimizing their code (similar to §2.3.4), is usually beneficial. On the contrary, WSC applications are well-known for their long tails and flat execution profiles [32], which are best addressed with scalable automatic optimization over many code locations.

Figure 2.3 shows that L1I cache misses in WSCs have similarly long tails. It plots the cumulative distributions of dynamic instructions, L1I, and L2-I misses over unique cache lines over a week of execution, fleetwide. Misses initially rise significantly steeper than instructions (inset), which suggests there are some pointwise manual optimizations with outsized performance gains. However, the distribution of misses tapers off, and addressing even two-thirds of dynamic misses requires transformations in $\approx 1M$ code locations, which is only conceivable with automation.

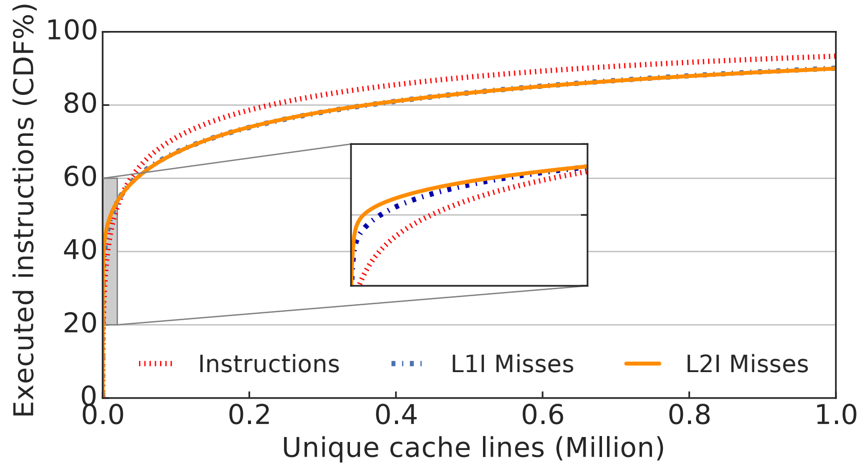


Figure 2.3: Fleet-wide distribution of executed instructions, L1-, and L2-instruction misses over unique cache lines. Like instructions, misses also follow a long tail.

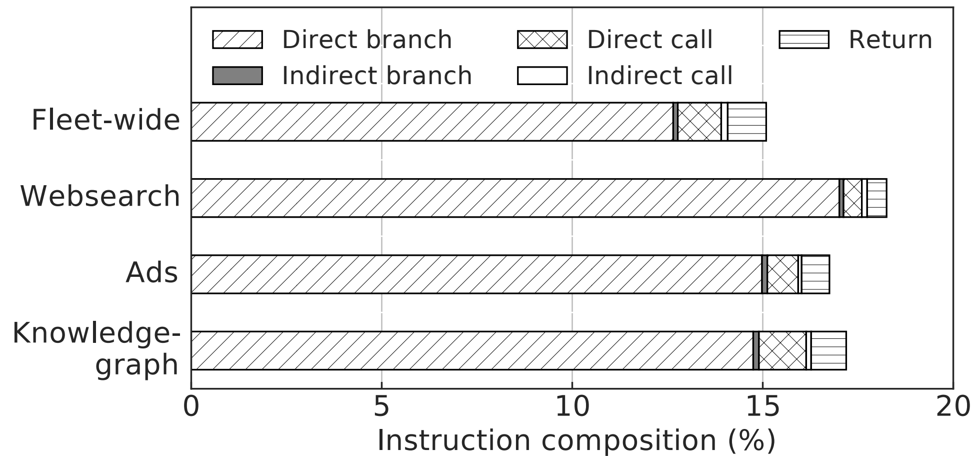


Figure 2.4: Control-flow instruction mixes for several WSC workloads. The remaining 80+% are sequential instructions.

2.2.2 Miss-Causing Instructions

When optimizing instruction cache misses, it is not only important to identify the instructions that miss themselves, but also the execution paths that lead to them. These are the *miss-causing* instructions.

In the vast majority of cases, the predecessor of a particular instruction in execution is simply the previous sequential instruction. Figure 2.4 illustrates this for several large WSC binaries, along with a fleetwide average from `AsmDB`. More than 80% of all executed

instructions are sequential (continuous or non-branching). The majority of the remainder (>10%) are direct branches, which are most typically associated with intra-function control-flow. Direct calls and returns, which jump into and out of functions, each represent only 1-2% of total execution. Indirect jumps and calls are even rarer.

Sequential instructions have high spatial locality and are thus inherently predictable. They can be trivially prefetched by a simple next-line prefetcher (NLP) in hardware. While there are no public details about instruction prefetching on current server processors, NLP is widely believed to be employed. And because NLP can virtually eliminate all cache misses for sequential instructions, it is the relatively small fraction of control-flow-changing instructions – branches, calls, and returns – which ultimately cause instruction cache misses and performance degradation.

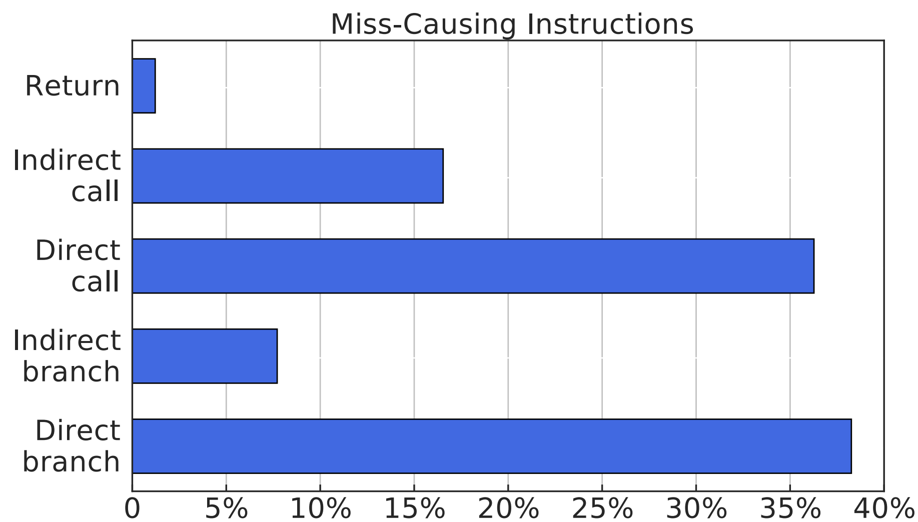


Figure 2.5: Instructions that lead to i-cache misses on a web search binary.

Intuitively, branches typically jump within the same function so their targets are more likely to be resident in the cache due to reuse (temporal locality) for backward-pointing branches such as loops, and either reuse or NLP (with sufficiently small target offsets) for forward-pointing branches. On the other hand, we expect calls to miss more often because they jump across functions which can span a wide range of distances in the address space. This defeats NLP and is more difficult to capture by reuse due to limited cache sizes and flat

callgraphs of WSC binaries.

In order to test this intuition, we send a full instruction trace from a web search binary through a simple L1 cache simulator with a next-two-line prefetcher, and mark each access that is a miss. For each miss, we look at the previous instruction in the program sequence which, assuming "perfect" (triggering on every access) NLP, is necessarily a control-flow-changing instruction. By counting these by type we built a profile of "miss-causing" instructions shown in the figure 2.5.

Discontinuity Type	Miss Percent	Miss Intensity
Direct Branch	38.26%	2.08
Indirect Branch	7.71%	59.30
Direct Call	36.27%	54.95
Indirect Call	16.54%	71.91
Return	1.22%	1.37

Table 2.1: Instruction miss intensities for web search

Interestingly, despite having higher temporal locality and being more amenable to NLP, direct branches are still responsible for 38% of all misses. These misses are comprised mostly of the small but long tail of direct branch targets that are greater than two cache lines away in the forward direction (18% of the profile). Perhaps more surprising is that direct calls account for 36% of all cache misses despite being only 1-2% of all executed instructions. This confirms that the probability of causing a miss is much higher for each call instruction, compared to that of a branch. Indirect calls, which are often used to implement function pointers and vtables, are even less frequently executed but also contribute significantly to misses. In contrast, returns rarely cause misses because they jump back to code that was often recently used. We summarize these probabilities in a dimensionless "miss intensity" metric, defined as the ratio of the number of caused misses to the execution counts for a particular instruction type. In other words, miss intensity helps us rank instruction classes by how likely they are to cause misses in the cache. We see from table 2.1 that the miss intensities of direct branches and returns are much lower than the other types, which indicates

their targets are more inherently cacheable.

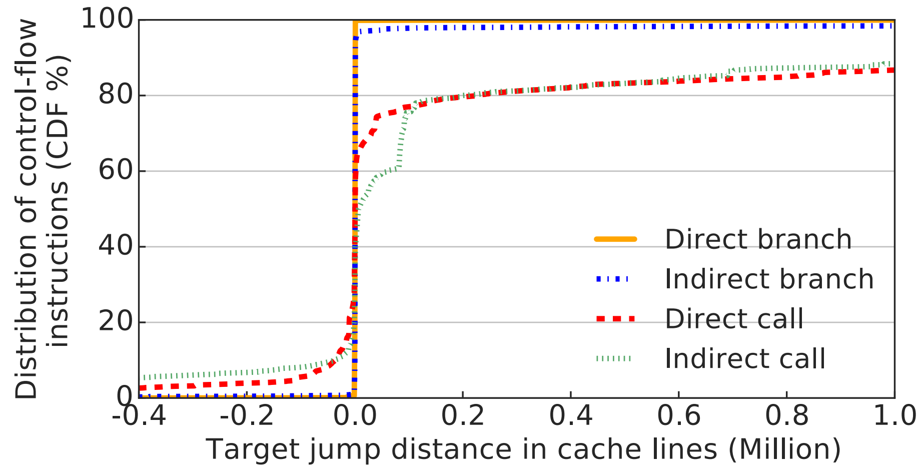


Figure 2.6: Fleetwide distribution of jump target distances.

We can confirm this is not limited to web search or due to the cache model used to tag misses by looking into fleetwide `AsmDB` jump distances. Figure 2.6 presents this data as a cumulative distribution function (CDF) of distances in bytes to the next instruction. Around 99% of all direct branches fleetwide jump to targets that are fewer than 500 cache lines away, and hence they are sharply centered around zero in the figure which is depicted at the scale of million cache lines. On the other hand, over 70% of direct calls have targets more than 100,000 cache lines (6.4 MB) away. While such distances do not guarantee a cache miss, they do increase the likelihood that simple prefetchers without knowledge of branch behavior will be insufficient to keep the data cache-resident.

Indirect calls and branches roughly track the behavior of their direct counterparts. However, they are so infrequent that their targets are relatively cold (and unlikely to be resident in the cache), leading to the high miss intensity in the table 2.1. Note that, in practice, indirect calls and branches tend to have a very small number of targets per instruction (figure 2.7 – 80+% of indirect calls and 58% of indirect branches always jump to a single address), which implies that they are very easily predictable with profile guided optimization.

In summary, the key takeaways from our instruction miss analysis are that 1) calls are the most significant cause of cache misses, and 2) branches with large offsets contribute

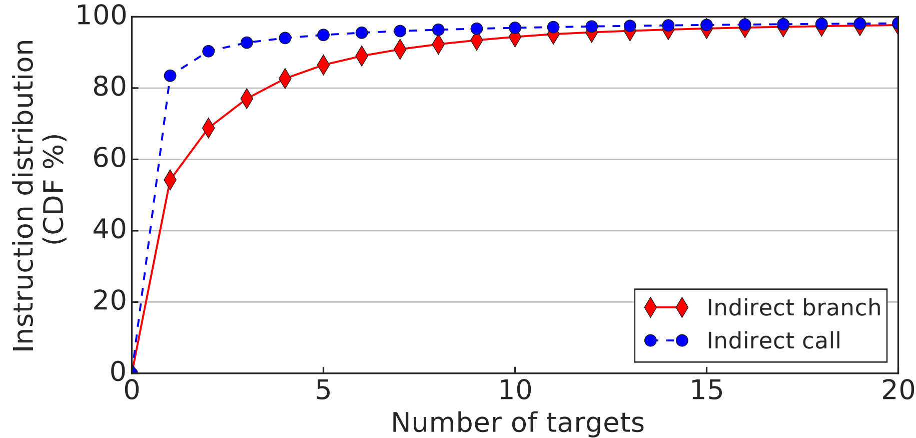


Figure 2.7: Cumulative distribution of number of targets for indirect jumps and calls.

significantly to misses because they are not fully covered by reuse and simple next-line prefetching.

2.3 Code Bloat and Fragmentation

In this section we outline some opportunities for improving L1I cache behavior at the granularity of groups of instructions. Namely, we identify reducing cache fragmentation on the intra-function and intra-cacheline level with feedback-driven code layout. A global database of assembly instructions and their frequencies such as `AsmDB` critically enables both prototyping and eventually productionizing such optimizations.

Briefly, fragmentation results in wasted limited cache resources when cold code is brought into the cache in addition to the necessary hot instructions. Such negative effects get increasingly prominent when functions frequently mix hot and cold regions of code. In this section we show that even among the hottest and most well-optimized functions in Google server fleet, more than 50% of code is completely cold. We attribute this to the deep inlining that the compiler needs to perform when optimizing typical WSC flat execution profiles. This suggests that combining inlining with more aggressive hot/cold code splitting can achieve better L1I cache utilization and free up scarce capacity.

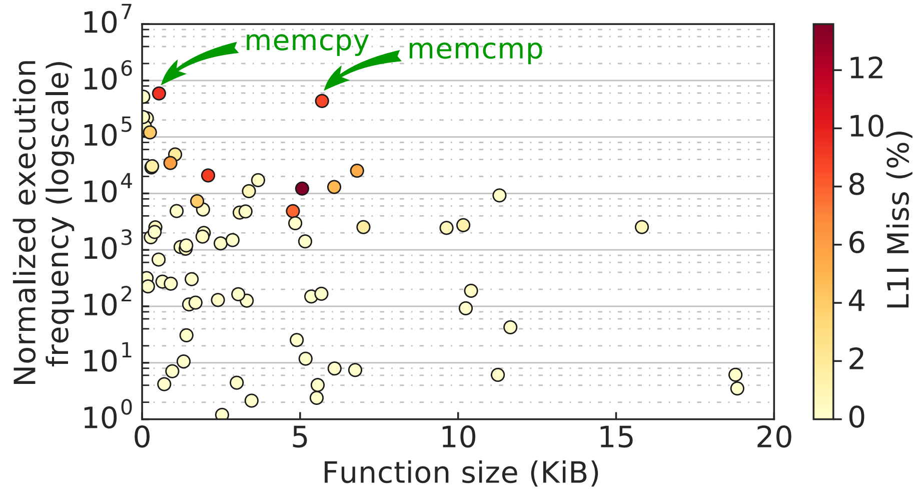


Figure 2.8: Normalized execution frequency vs. function size for the top 100 hottest fleetwide functions. `memcmp` is a clear outlier.

2.3.1 Code Bloat

One common symptom for excessive L1I cache pressure is code bloat, or unnecessary complexity, especially in frequently-executed code. Figure 2.8 is an attempt to diagnose bloat from `AsmDB` data – it plots normalized function hotness (how often a particular function is called over a fixed period) versus the function's size in bytes for the 100 hottest functions in Google WSCs. Perhaps unsurprisingly, it shows a loose negative correlation: Smaller functions are called more frequently. It also corroborates prior findings that low-level library functions ("datacenter tax" [32]), and specifically `memcpy` and `memcmp` (which copy and compare two byte arrays, respectively) are among the hottest in the workloads we examined.

However, despite smaller functions being significantly more frequent, they are not the major source of L1I cache misses. Overlaying miss profiles from GWP onto the figure 2.8 (shading), we notice that most observed cache misses lie in functions larger than 1 KB in code size, with over half in functions larger than 5 KB.

This contradicts traditional optimization rules of thumb, like "Most [relevant] functions are small". But it also holds for execution cycles – as illustrated in the figure 2.9 – only 31%

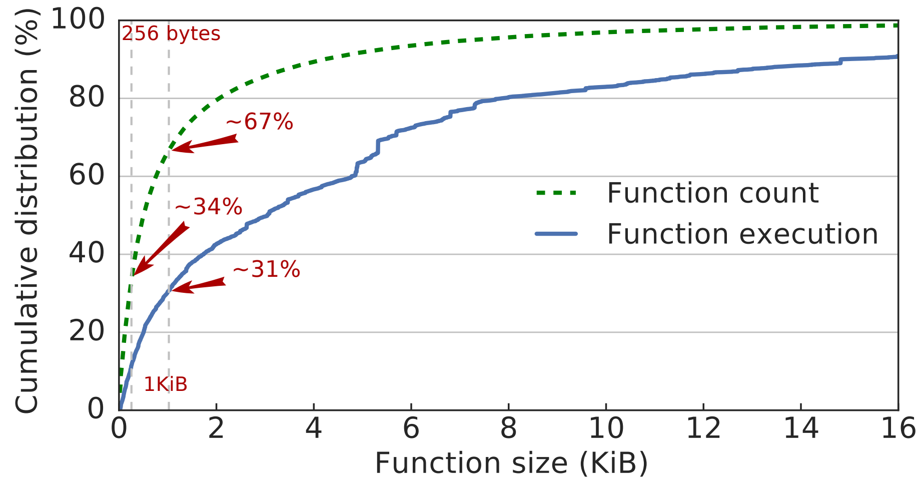


Figure 2.9: Distribution of execution over function size.

of execution fleetwide is in functions smaller than 1 KB. Small functions are still prevalent: 67% of all observed functions by count are smaller than 1 KB. However, a large portion of them are very cold. This suggests that, as expected for performance, small hot functions get frequently inlined with the help of profile feedback[1].

The ubiquity and overall aggressiveness of inlining is best illustrated in the figure 2.10, which plots the depth of observed inline stacks over the 100 hottest functions. Most functions 5 KB or larger have inlined children more than 10 layers deep. While deep inlining is crucial for performance in workloads with flat callgraphs, it brings in exponentially more code at each inline level, not all of which is necessarily hot. This can cause fragmentation and suboptimal utilization of the available L1I cache.

2.3.2 Intra-Function Fragmentation

In order to understand the magnitude of the potential problem, we quantify code fragmentation on the function level. We more formally define fragmentation to be the fraction of code that is definitely cold, that is the amount of code (in bytes) necessary to cover the last 10%, 1%, or 0.1% of execution of a function. Because functions are sequentially laid out in

[1] At the time of collection, over 50% of fleet samples were built with some flavor of profile-guided optimization.

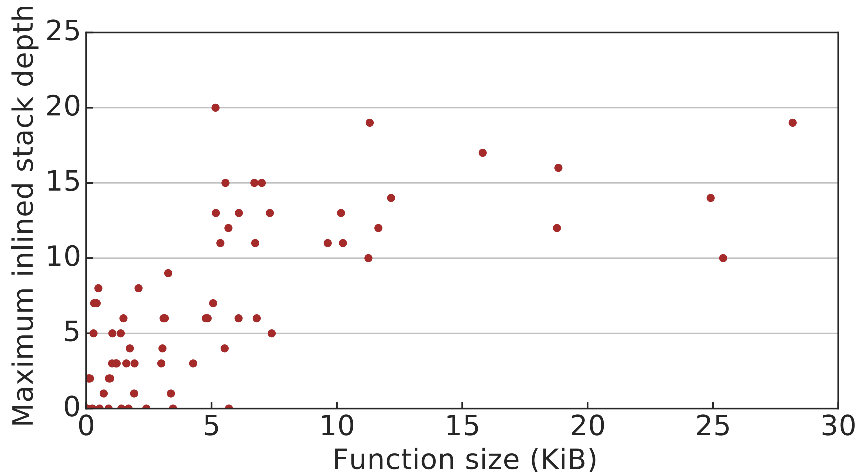


Figure 2.10: Maximum inlining depth versus function size for the 100 hottest fleetwide functions.

memory, these cold bytes are very likely to be brought into the cache by next-line prefetching. Intuitively, this definition measures the fraction of L1I cache capacity potentially wasted by bringing them in.

Using AsmDB data, we calculate this measure for the top 100 functions by execution counts in Google sever fleet. Figure 2.11 plots it against the containing function size. If we consider code covering the last 1% of execution "cold", 66 functions out of the 100 are comprised of more than 50% cold code. Even with a stricter definition of cold ($< 0.1\%$), 46 functions have more than 50% cold code. Perhaps not surprisingly, there is a loose correlation with function size – larger (more complex) functions tend to have a larger fraction of cold code. Generally, in roughly half of even the hottest functions, more than half of the code bytes are practically never executed, but likely to be in the cache.

2.3.3 Intra-Cacheline Fragmentation

Fragmentation in the L1I cache also manifests itself at an even finer granularity – for the bytes within each individual cacheline. Unlike cold cache lines within a function, cold bytes in a cache line are always brought in along the hot ones, and present a more severe performance problem. We defined a similar metric to quantify intra-cacheline fragmentation:

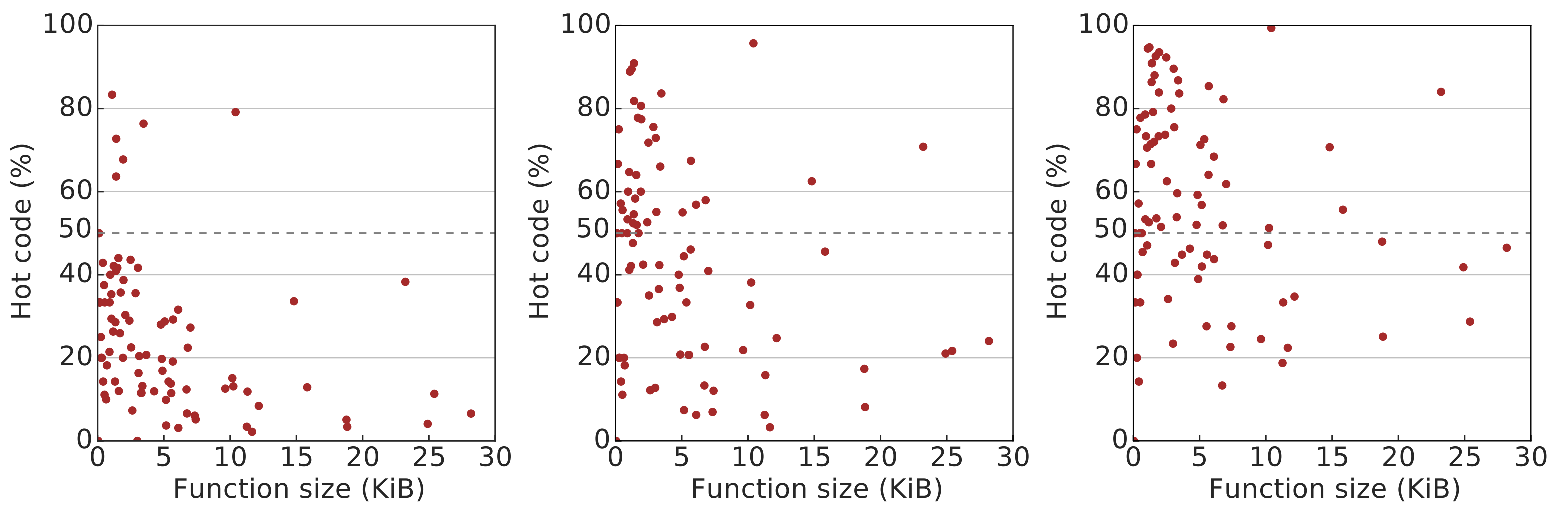


Figure 2.11: Fraction of hot code within a function among the 100 hottest fleetwide functions. From left to right, “hot code” defined as covering 90%, 99% and 99.9% of execution.

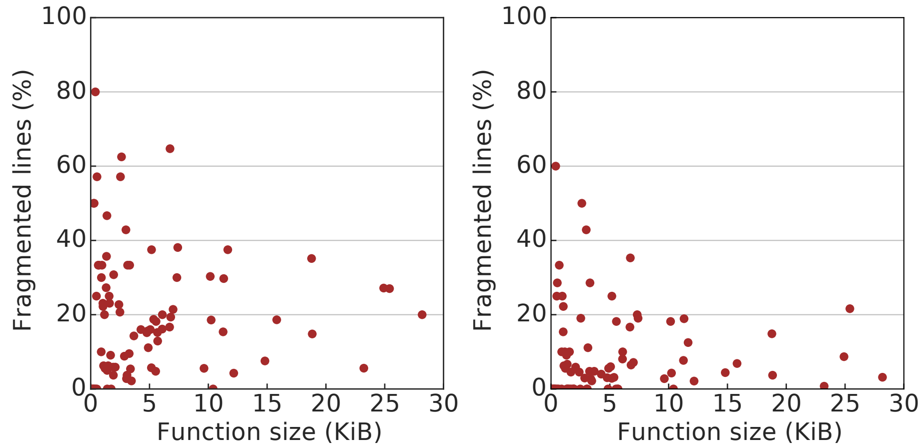


Figure 2.12: Intra-cacheline fragmentation vs function size for hotness thresholds of 90%, and 99%.

counting the number of bytes (out of 64) necessary to cover 90% or 99% of the line's accesses. Similarly to the last section, we declare a line fragmented if it only uses 50% or fewer of its bytes to cover execution. Figure 2.12 shows the fraction of fragmented lines for each of the top 100 functions in the Google server fleet. At least 10% of the functions have more than 20% of the cache lines that are fragmented, and fragmentation is more common for small functions. In other words, while these functions are executing, at least 10% of L1I cache capacity is stranded by fragmented lines. This suggests opportunities in basic-block layout, perhaps at link, or post-link time, when compiler profile information is precise enough to reason about specific cache lines.

2.3.4 Memcmp and the Perils of Micro-Optimization

To illustrate the potential gains from more aggressive layout optimization, we focus on the most extreme case of bloat we observed in `AsmDB` – the library function `memcmp`.

`memcmp` clearly stands out of the correlation between call frequency and function size in the figure 2.8. It is both extremely frequent, and, at almost 6 KB of code, $10\times$ larger than `memcpy` which is conceptually of similar complexity. Examining its layout and execution patterns (figure 2.13) suggests that it does suffer from the high amount of fragmentation we

observed fleetwide in the previous section. While covering 90% of executed instructions in `memcmp` only requires two cache lines, getting up to 99% coverage requires 41 lines, or 2.6 KB of cache capacity. Not only is more than 50% of code cold, but it is also interspersed between the relatively hot regions, and likely unnecessarily brought in by prefetchers. Such bloat is costly – performance counter data collected by GWP indicates that 8.2% of all L1I cache misses among the 100 hottest functions are from `memcmp` alone.

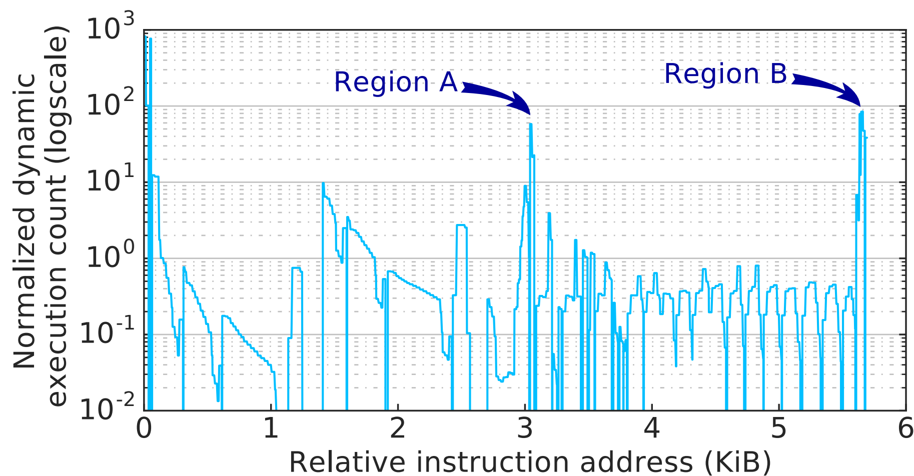


Figure 2.13: Instruction execution profile for `memcmp`. 90% of dynamic instructions are contained in 2 cache lines; covering 99% of instructions requires 41 L1I cache lines.

A closer look at the actual code from glibc can explain the execution patterns in the figure 2.13. It is hand-written in assembly and precompiled, with extensive manual loop unrolling, many conditional cases for the various alignments of the two source arrays, and large jump tables.

In our experience, code usually evolves into similar state from over-reliance on micro-optimization and micro-benchmarking. While writing in assembly can in rare cases be a necessary evil, it prevents the compiler from doing even the most basic feedback-directed code layout optimizations. For example, it cannot duplicate or move the “compare remainders” and “exit” basic blocks marked `RegionA` and `RegionB` in the figure 2.13 closer to the cases that happen to call them the most (in this case the beginning of the function). This results in expensive and hard-to-prefetch jumps, and cache pollution. Similarly, when

mostly evaluated on microbenchmarks, all relevant code usually fits in the L1I cache, which is certainly not the case for large applications. This encourages developers to add more elaborate corner cases (e.g. for alignment) that improve the microbenchmark without regard to bloat.

We tested this hypothesis by macro-benchmarking a version of `memcmp` that is specifically optimized for code size (only 2 L1I cache lines) and locality. In short, it only special-cases very small string sizes (to aid the compiler in inlining very fast cases) and falls back to `rep cmps` for larger compares. Even though it achieves slightly lower throughput numbers than the glibc version in micro-benchmarks, this simple proof-of-concept showed an overall 0.5%-1% end-to-end performance improvement on large-footprint workloads like web search.

Interestingly, a more recent related work [11] has performed a similar in-depth analysis on few glibc functions including memcmp. The authors in [11] also have a similar observation that handwritten assembly, especially in functions like the ones in glibc have huge performance cost on a WSC. They have shown that, an alternative implementation in a high-level language and usage of the extensive feedback directed compiler optimizations increased the performance of the fleet by 1%.

`AsmDB` allows us to spot extreme cases such as `memcmp` over thousands of applications. In this case, `memcmp` was the single immediately apparent outlier both in terms of code bloat and L1I cache footprint. Manually optimizing it for code size was practical and immediately beneficial.

However, manual layout optimization does not scale past extreme outliers. Generalizing similar gains is in the domain of compilers. Compiler optimizations like hot/cold splitting [14] and partial inlining [64] aim to address fragmentation by only inlining the hot basic blocks of a function. However, they have recently been shown to be particularly susceptible to the accuracy of feedback profiles [50], especially with sampling approaches like AutoFDO [12].

The high degree of fragmentation we observed suggests there is significant opportunity to improve L1I cache utilization by more aggressive and more precise versions of these optimizations than found in today's compilers. Alternatively, post-link optimization could be a viable option which does not suffer from profile accuracy loss. The latter approach has been shown to speed up some large datacenter applications by 2-8% [50].

Chapter 3

Instruction Cache Miss Cost in Modern Processors

Caches play a vital role in improving processor performance. However, caches are already resource-constrained and will never have enough capacity to accommodate today's faster-growing workloads. Chapter 2 detailed the root-causes for the L1I misses at the scale of a WSC. To further understand the problem without revealing Google proprietary information, we scale the problem to manifest in the same way on the well-known 445.gobmk SPEC Benchmark. As described later in §4.3.4, 445.gobmk is one of the SPEC programs with large working set size.

Accurately scaling the WSC problem to a SPEC benchmark program means using a smaller sized L1I Cache of 8kB with 4-way set-associative and applying tools like BOLT [50] from Facebook to the code to minimize conflict misses and to maximize the potential of Next-Line Prefetch (NLP) hardware that is believed to exist in the modern processors. We applied BOLT's Basic Block Layout Optimization pass to 445.gobmk, which organizes the code based on profile information to minimize the amount of taken branch instructions. By doing so, the program flow tends to be sequential in most cases, and further benefits from NLP.

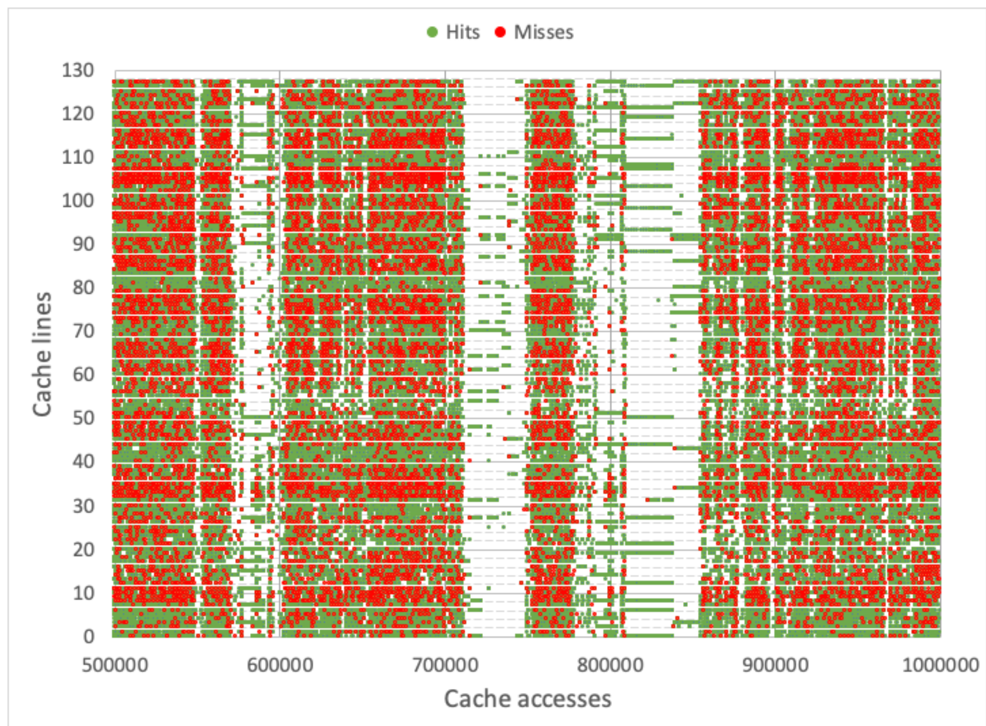
Our simulations in this case use Zsim [58] with careful attention to prefetch timing. Figure 3.1 shows the distribution of instruction cache accesses over cache lines for a region of 445.gobmk execution. Each green dot in the plot represents an L1I Cache hit, and each red dot represents a miss. The top figure 3.1a presents the region when just NLP is configured, whereas the bottom figure 3.1b presents the same region when Facebook BOLT is applied to the program and the NLP is configured. BOLT combined with NLP's effect is evident in the figure 3.1b, where there are fewer misses than with purely NLP in figure 3.1a. However, even with NLP configured as found in Intel processors and with the use of the best available software tools (e.g., Facebook's BOLT), this figure reveals a disturbing picture. This behavior is prevalent in many WSC and other types of workloads. This experiment yields 8.8 L1I Cache Misses Per Thousand Instructions (MPKI) even with nearly 30 million prefetches.

Additionally, figure 3.1 doesn't show any clustering of misses, they are distributed across all the lines over the entire time duration. If there were fewer addresses responsible for a majority of misses, they could have been handled by observing the pattern. Since that is not the case, only option seems to be some form of automation with a smarter replacement policy.

3.1 Replacement Policies Dictate Cache Performance

For many decades, work has focused on improving processor performance by reducing cache misses [3,6,26,28,37,38,43,48,52,55]. Apart from NLP and code optimizations, another important way to reduce cache misses for a given cache size, line size, and associativity is to improve the cache replacement policy. The classic LRU (least recently used) replacement policy exploits temporal locality by evicting the line least recently accessed [51,66]. Bélády's OPT algorithm achieves the minimum number of misses through ideal cache replacement, but it is unrealizable in practice because it requires perfect knowledge of future references [8].

(a) I-Cache behavior with NLP



(b) I-Cache behavior with Facebook BOLT and NLP

Figure 3.1: L1I cache accesses in 445.gobmk benchmark on hardware scaled to match WSC behavior.

OPT's value is not just theoretical, however, because it informs the design of many practical cache replacement policies [26,28,43,52]. These and almost all previously proposed cache replacement policies aim to *minimize misses* using only *reference information*.

3.2 Prior Cost-Aware Replacement Policies for Data Cache

Architects have long recognized that not all cache misses have the same cost in modern processors owing to several latency tolerating capabilities [29,30,35,47,53,63]. Cost-aware cache replacement policies recognize this by attempting to increase performance even at the cost of increased cache misses. For modern architectures, many misses do not impact performance at all. For example, an aggressive out-of-order processor can entirely tolerate certain first-level data cache (L1D) misses without any negative performance impact [39]. Likewise, many modern processors have decoupled front-ends with early fetch engines that can tolerate certain first-level instruction cache (L1I) misses without causing decode starvation, a state in which the head instruction of the instruction queue feeding the decode stage is not yet available [24,25,54].

The optimal cost-aware replacement policy (CSOPT) is the perfect-knowledge cost-aware cache replacement algorithm [30]. Unfortunately, CSOPT, like OPT, is unrealizable in practice. Prior research has produced realizable cost-aware data cache replacement policies [35, 47, 53, 63]. For example, the MLP-aware Linear (LIN) policy and related techniques prioritize lines that miss with lower memory-level parallelism (MLP) [53]. Other techniques consider load criticality as approximated by a variety of methods [35,47,63].

We are unaware of prior work on cost-aware replacement policies for instruction caches. This is partly owing to the fact that instruction cache had not gained attention from researchers until recently.

3.3 Why is Instruction Cache Miss Cost Important?

Modern processors encompass buffers between pipeline stages to increase efficiency. The buffer between the fetch and decode stage, is referred to as decode queue in this context. It acts as a pool that holds on to the bytes until the decode stage is ready to decode them sequentially, as the front-end primarily processes instructions in-order as they occur in the program context.

Instruction fetch is responsible for keeping the decode stage and in-turn the processor pipeline fed. If the processor could perfectly predict the future direction and target of every control-flow instruction, instruction fetch could issue all of its memory requests early enough to tolerate the latency to main memory without starving decode (the state wherein the decode queue is empty and as a result decode stage is unable to process any instruction even if it has the bandwidth to do so). Unfortunately, even the best branch predictors are not perfect. They are, however, quite good. Modern processor front-ends incorporate decoupled, aggressive fetch engines guided by excellent branch predictors, large Branch Target Buffers (BTBs), and predecoders [25]. Such front-ends are capable of accurately fetching several tens or even hundreds of instructions early. Instruction decode queues filled this way can often tolerate L1I misses before emptying and leading to decode starvation. This is especially true when the L1I miss is served by L2 cache.

Branch mispredictions invalidate early fetch work, requiring the flush of the instruction queue. Re-steering the front-end takes time, and more time is necessary for fetch to run far enough ahead of decode to fill the instruction decode queue enough to tolerate L1I misses. This concept suggests a cache replacement policy based on proximity to poorly predicted branch targets. Our early explorations in this direction considered cache replacement policies that were either too complex, ineffective, or both. Part of the reason for this is that not all branch mispredictions lead to decode starvation. Often times the lines necessary after re-steer are always in L1I despite the branch mispredict. For example, this is the case for near-target branches in which the mispredicted path and the committed path share the same

L1I cache lines. The mispredicted path fetch acts as a prefetch in such a scenario.

Beyond branch prediction, there are many factors that interact to determine whether or not an L1I miss will cause decode starvation. For example, a *stalled* decode cannot *starve*. Decode stalls occur when the processor backend cannot accept more instructions. When decode stalls, it does not attempt to pull from the instruction queue, a necessary condition for starvation. For simplicity reasons, we did not consider integrating the state of the backend or other component factors into the design.

3.4 Decode Starvation as a Cost Metric

While predicting starvation by its component factors is hard, observing starvation during execution is easy. Existing signals assert when decode starvation occurs. Further, when decode starvation occurs, the address for which decode is waiting is also tracked. The processor knows this information many cycles before the respective line that missed is inserted into the cache. This information is available well in advance, but that does not necessarily mean it is of value to a cost-aware cache replacement policy.

Fig. 3.2 shows the percentage of L1I cache misses that cause decode starvation (first bar). This figure shows that most of the L1I cache misses cause decode starvation, but a full quarter of L1I misses do not. Since many misses starve while many others do not, this suggests that a simple starvation signal may be sufficient. If, for example, all or almost all misses starved, a cost-aware cache replacement policy might need to calculate the time in cycles of each starvation, leading to added complexity.

Fig. 3.2 also shows the distribution of L1I cache misses according to the decode starvation rate of their addresses (second bar). The figure shows that more than 75% of the L1I misses that cause decode starvation involve “starvation-prone” addresses (>90% of their misses cause decode starvation). This suggests that selecting these lines for high-priority treatment in a cost-aware cache replacement policy could avoid a sizable fraction of total starvations.

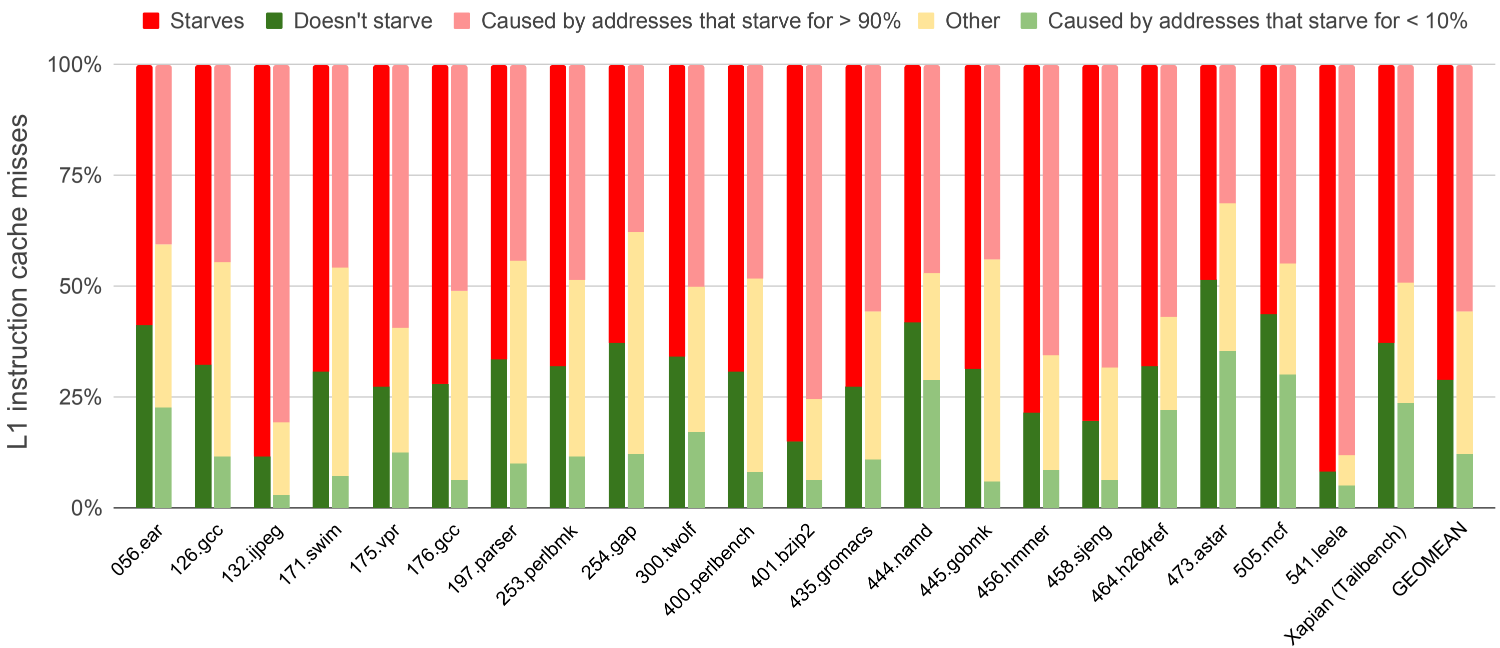


Figure 3.2: Percentage of L1I misses causing decode starvation (first bar) and the distribution of L1I misses according to the decode starvation rate of their addresses (second bar), when run on a system with an 8kB L1I. (§ 4.3 details the simulator and hardware configuration.)

It also suggests that these lines are easy to find as they almost always starve.

Zooming in on 445.gobmk in fig. 3.2, there exists a similar pattern where nearly 30% of the L1I cache misses never cause decode starvation. Put in other words, they never impact the processor performance. Looking back into the cache access pattern of 445.gobmk in fig. 3.1, although there exists non-trivial amount of misses, the replacement policy should only consider to reduce those misses that impact the performance. Next chapter 4 covers the cost-aware instruction cache replacement policy in detail.

Chapter 4

Cost-Aware Instruction Cache Replacement Policy

Modern processors' frontend have evolved to tolerate memory latencies. As a result not all cache misses have the same performance impact, which are described in detail in chapter 3. However, to-date there is no instruction cache replacement policy proposed, that prioritizes the cost of the miss. To address the lack of cost-aware cache replacement policies for instruction caches, this work presents EMISSARY (Enhanced MISS-Awareness Replacement Policy).

Observing that modern architectures completely tolerate many L1I misses, keeping lines with tolerated L1I misses in the L1I cache has less utility than keeping lines in the L1I whose misses cause decode starvation. The key to making this work involves segregating tolerated L1I misses from those that cause starvation. EMISSARY prioritizes inserted lines whose miss caused decode starvation over those whose miss did not.

Without the need to track history, to coordinate with prefetchers, to make predictions, or to perform complex calculations, EMISSARY consistently improves performance and saves energy while remaining simple. In a few cases, EMISSARY configurations outperform Bélády's OPT, the clairvoyant minimal-miss replacement policy not realizable in practice.

The outline of this chapter is as follows.

- §4.1 provides an overview tour of the EMISSARY techniques. Since EMISSARY is a bimodal technique (misses are treated in one of two ways), this overview is set in the context of prior bimodal techniques. This section also highlights the value of persisting this bimodality over a line's entire lifetime in the cache, a feature we believe to be an EMISSARY first.
- Using the observation that bimodal selection and treatment are orthogonal, §4.2 introduces a notation covering the space of EMISSARY techniques and related prior works. §4.2 also describes the EMISSARY algorithm.
- §4.3 outlines how and by how much EMISSARY outperforms related and prior work in both speed and energy. In short, a simple EMISSARY configuration yields geomean speedups of 2.1% for an 8K L1I, 2.6% for 16K, and 1.5% for 32K on the set of programs with an LRU MPKI (misses per kilo instructions) greater than 1.0. (Cache replacement policies have little impact in cases without many misses.) With little added complexity, EMISSARY saves a corresponding amount of energy. These numbers are significant for the modeled modern architectures given how well they fetch instructions early guided by sophisticated branch predictors and how often they tolerate L1I misses when they do occur.
- Using 400.perlbench as a model for other programs expressing the same interesting behaviors, §4.5 explores, among other things, how EMISSARY can improve performance even when the number of decode starvations (not just misses) *increases*.
- §4.6 describes related works in more detail.

4.1 An Overview Tour of EMISSARY

This Dissertation advocates treating L1I cache misses that starve decode differently from those that do not. Since starvation occurs when the head instruction of the instruction queue feeding decode is not yet available, observing decode starvation is simple. Thus, the challenge lies chiefly in choosing the *bimodal selection policy* (e.g., select by starvation history or a single occurrence?), the *bimodal treatment policy* (i.e., what to do when a line is selected?), and other elements to improve performance and energy while keeping complexity low. This section provides an overview tour of these elements using Fig. 4.1 to illustrate their impact on the performance, MPKI, and commit-path decode starvations of the 400.perlbench benchmark. Fig. 4.1 reports commit-path starvations because off-path starvations generally do not impact performance.

One previously proposed mechanism for realizing bimodal treatment is an *insertion policy* in an LRU *replacement policy* cache that inserts low-priority lines in the LRU position instead of the default MRU (most recently used) position [52]. The *MRU Insert:Starvation* policy combines this bimodal treatment with the direct use of the decode starvation signal for the bimodal selection. As shown in Fig. 4.1, *MRU Insert:Starvation* outperforms LRU in performance, MPKI, and decode starvation cycle count. To test the value of starvation as a useful signal for bimodal selection, the miss starvation signal is replaced with a random signal with probability 1/32 in *MRU Insert:Random (called BIP in [52])*. Interestingly, the performance, MPKI, and starvation count are relatively unchanged by this replacement. [1] While this result suggests that decode starvation is not a valuable bimodal selection signal, as described next, the reason is actually that this bimodal treatment mechanism does not materialize the impact long enough. Specifically, after insertion, only a few references are enough to erase any differentiation obtained by MRU/LRU position selection.

To realize a lasting effect of the starvation signal, EMISSARY cache replacement policies

[1] *MRU Insert:Random (BIP)* performs so well vs. LRU because most (31/32) single-reference lines get evicted first when inserted in the LRU position while lines that miss frequently have more opportunities (1/32 *per miss*) to be selected for the MRU position [52].

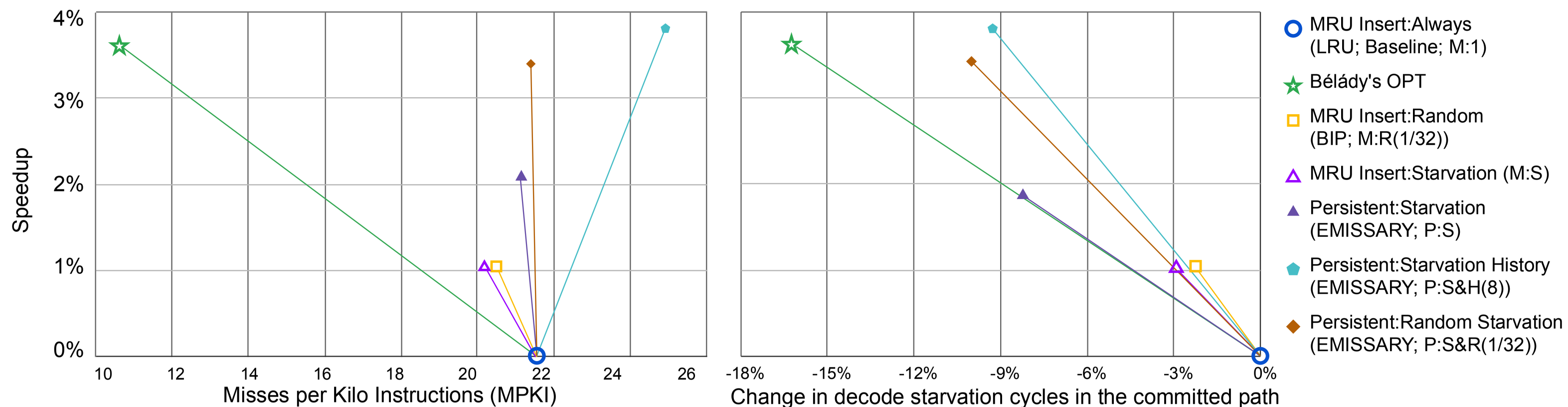


Figure 4.1: Performance, misses, and commit-path decode starvations of various cache replacement policies for 400.perlbench on a 32kB 8-way L1I cache. (§4.3 details the simulator and hardware configuration. §4.2 describes the notation in parentheses.)

are *persistently* bimodal. Instead of differentiating solely by insertion policy, an EMISSARY replacement policy treats selected lines differently for the *entire duration of their time in the cache*. They do this, in part, by not allowing low-priority line insertion (e.g., from a miss without decode starvation) to evict any of up to N protected high-priority lines in the set. Only high-priority lines (e.g., from a miss with decode starvation) may be protected in this way, but not all high-priority lines are protected (i.e., when more than N lines in a set are high-priority). EMISSARY marks each line in the cache with a priority bit, setting $P = 1$ if high-priority. This bit does not change while the line remains in the cache. §4.2 contains a complete description of EMISSARY's mechanism. The EMISSARY configurations in Fig. 4.1 support up to six protected lines ($N = 6$) per set, leaving two lines in the 8-way cache available for low-priority misses. Having low-priority lines bypass the cache was not found to be effective, so all misses in EMISSARY result in an insertion.

Observe the difference between *MRU Insert:Starvation* and *Persistent:Starvation* in Fig. 4.1. Using the EMISSARY replacement policy (*Persistent*) instead of the *MRU Insert* policy produces a significant reduction in decode starvations on the committed path and a similarly significant increase in performance. Interestingly, *Persistent:Starvation* has a higher MPKI than *MRU Insert:Starvation*, despite its increased performance. (More on that below.)

A line with a lower fraction of starvations (or no starvations) among its recent misses should be given lower priority than a line that starves decode on every recent miss. The *Persistent:Starvation History* policy implements this distinction with a bimodal selection that marks a line that starves as high-priority only if *all* 8 of its most recent prior misses also starved. As shown in Fig. 4.1, *Persistent:Starvation History*, this more selective, history-aware policy significantly outperforms *Persistent:Starvation*. Interestingly, *Persistent:Starvation History*, a realizable cache replacement algorithm, also outperforms Bélády's OPT. This single experiment is an instance proof that minimizing L1I misses does not necessarily produce the best performing result on a modern architecture. Put another way, some

misses are actually desirable and necessary to increase performance further. Missing on lines that do not cause decode starvation makes room in the cache for those that do.

A technique that approximates *Persistent:Starvation History* at a much lower cost is *Persistent:Starvation Random*. *Persistent:Starvation Random* only gives higher priority to misses that starve with random probability 1/32. As others have observed, random selection on an event can have an effect similar to counting the history of that event [52]. As shown in Fig. 4.1, *Persistent:Starvation Random* does nearly as well as *Persistent:Starvation History*. This small degradation in performance comes with an impressive reduction in misses and complexity. To track recent starvations, *Persistent:Starvation History* needs 3 bits for each entry in the *L2 cache* to record the starvation history of lines that repeatedly miss in L1. This is a significant addition over the single bit per L1I line required by the EMISSARY techniques promoted by this work.

As shown by the lines connecting the baseline (LRU) to each configuration in Fig. 4.1, starvation count on the committed path is better correlated to performance than MPKI. However, this correlation does not always hold. *Persistent:Starvation History* has more starvations than *Persistent:Starvation Random* but higher performance. As explained in §4.5, the distribution of starvations is of importance as well because distributed decode starvations can be tolerated better by the processor's back end than those that occur in groups. Even when EMISSARY increases starvation counts, it tends to perform better by distributing them more uniformly, reducing stress on the latency tolerance mechanisms later in the pipeline.

4.2 The EMISSARY Policies

EMISSARY instruction cache replacement policies build on the lessons of chapter §3, including that lines that caused performance issues will tend to do so while those that did not will tend not to do so. In L1I, an EMISSARY cache leverages this by holding on to starvation-causing lines longer, even if it is less recently accessed than other, starvation-free

Notation	Description
1	Always High-Priority
0	Never High-Priority
R(r)	High-Priority with random probability r
S	High-Priority when a line's miss causes starvation
H(n)	High-Priority when a line's n-prior misses caused starvation

Table 4.1: Mode Selection Options

lines. Thus, EMISSARY cache replacement policies are bimodal. Bimodal techniques are described using two orthogonal aspects: *mode selection* and *mode treatment*. These are described next.

4.2.1 Mode Selection

For discussion, *high-priority* and *low-priority* are the names given to the two modes of a bimodal cache replacement policy. Table 4.1 shows the mode selection options for the space of realizable cache replacement algorithms referenced in this chapter. These mode selection options can be combined in Boolean equations. For example, S&R(1/32) requires a missed line both to have caused starvation during the miss AND to have been lucky enough to have rolled that one magic number on a 32-sided die. Because H(n) is needlessly complex, EMISSARY policies all contain S in their mode selection equations. For all policies in this chapter, mode selection is determined once at cache line insertion. LRU can be thought of as a bimodal predictor degenerated to treat all inserted lines as high-priority (i.e., 1, Always High-Priority), hence the placement in the MRU position.

4.2.2 Mode Treatment

A meaningful bimodal cache replacement policy must treat a line differently based upon the mode selected. Thus, the next aspect is how to treat high-priority lines differently from low-priority lines. All realizable cache replacement policies in this paper use one of the two

Notation	Description
M	Insert High-Priority lines as MRU, otherwise LRU
P(N)	Protect up to N MRU High-Priority lines from eviction

Table 4.2: Mode Treatment Options

Algorithm 1: The EMISSARY Eviction Policy

```
1 if number of high-priority (P = 1) lines <= N then
2 |   Evict the LRU among the low-priority (P = 0) lines
3 else
4 |_  Evict the LRU among all lines
```

bimodal behaviors shown in Table 4.2.

The first, M, assumes an LRU cache. Bimodality comes from inserting high-priority lines into the cache's MRU position while placing low-priority lines into the cache's LRU position. This mechanism is studied in prior work [52].

The second, P(N), is the EMISSARY behavior. It is described by Algorithm 1. P(N) techniques do not act on priority at insertion. Instead, the priority is recorded as a P-bit associated with each line, and this priority impacts eviction. High-priority lines have $P = 1$, while low-priority lines have $P = 0$. When inserting a line into a P(N) cache, if the number of high-priority lines is less than or equal to the max (N), the line to be inserted (regardless of its priority) replaces the LRU of the low-priority lines. Thus, this step in line 2 may increase the number of high-priority lines in the cache but cannot reduce it. For insertions where the number of high-priority lines (N) is greater than N, the cache evicts the LRU of all lines without risk replacing one of the N MRU high-priority lines. Note that an EMISSARY cache cannot have fewer than N high-priority lines at any point after reaching that many. (Experiments with configurations that periodically reset the P-bits demonstrated an inconsistent performance impact regardless of period length.)

To some degree, the EMISSARY treatment option is orthogonal to the LRU or pseudo-LRU (PLRU) algorithm used. For line 4 of Algorithm 1, true LRU, tree-PLRU, bit-PLRU, or any other LRU algorithm would be appropriate. Line 2 of Algorithm 1 requires consideration

because it requires the calculation of the LRU for a subset of the lines (i.e., the low-priority lines). For a true LRU, no additional work is necessary on reference updates. There is, however, additional work of calculating the line to evict in the form of finding the first LRU line not marked with $P = 1$. There are more efficient options with PLRU implementations, but the easiest to describe here is an extension of bit-PLRU [49]. In bit-PLRU, each line has an associated status bit that is set to 1 on the reference. When the last 0 bit would be set to 1, all others are reset to 0. The LRU for replacement is the first line, in some order, with a 0. For calculating the LRU of a changing subset of the lines (i.e., the low-priority lines), the P bits serve as a mask excluding high-priority lines from becoming 0. This prevents high-priority lines from being selected at replacement, and it prevents them from being considered in the PLRU process at reference. With this mechanism, EMISSARY adds two bits of state per line to the baseline cache (the P bits and the bit-PLRU bits). For the purposes of this paper, we model EMISSARY with LRU.

4.2.3 Cache Replacement Policies

Table 4.3 shows the prior work and proposed realizable cache replacement policies used in this work. Each policy is a combination of a mode selection option (individually or by combination with a Boolean expression) and a mode treatment option described earlier.

4.3 Experimental Exploration

This section first describes the machine model and simulation infrastructure. Then, it outlines the benchmarks selected. Using those programs, it presents the selection of the subspace of cache replacement configurations for measurement. Finally, using simulation results for these programs and configurations, it highlights performance and energy trends.

Notation	Description
M:1	Always insert in MRU position; Classic LRU; Baseline
M:0	Never insert in MRU position (only LRU position); LRU Insertion Policy (LIP) [52]
M:R(r)	MRU position insertion with random probability r; Bimodal Insertion Policy (BIP) [52]
M:S	MRU position insertion when starvation occurs
M:S&R(r)	MRU position insertion when starvation occurs along with random probability r
P(n):R(r)	EMISSARY bimodal behavior only; high-priority lines selected with random probability r (not an EMISSARY selection)
P(n):S	An EMISSARY policy; high-priority selection based on starvation
P(n):S&R(r)	An EMISSARY policy; high-priority selection based on starvation along with random probability r
P(n):S&H(h)	An EMISSARY policy used in §4.1; high-priority selection based on starvation along with an unbroken history of h starvations

Table 4.3: Realizable cache replacement policies

4.3.1 Machine Model and Simulator

This study of cache replacement policies uses gem5, a popular cycle-accurate simulator [9], running in a detailed CPU model in SE (System Emulation) mode. Table 4.4 shows the machine model parameters used. The L1I sizes simulated include 32kB, 16kB, and 8kB. All caches are 8-way to reflect the associativity found in modern Intel x86 processors. The true LRU replacement policy serves as the universal baseline.

For all reference simulations (i.e., those producing results presented in §4.3.4 and §4.3.5), we used the Intel "Skylake-like" model. All reference simulations include a fast-forward of 1 billion instructions followed by a detailed simulation of 250 million instructions. Since the starvation signal is not available during fast-forwarding, fast-forwarding for all configurations is done with LRU cache updates. For all exploration and presented deeper explorations (i.e., those presented in §3.3, §4.1, §4.3.3, and §4.5), we used the "Default gem5" model, and performed just the detailed simulations of the first 100 million instructions, without any fast-forward.

Field \ Model	Default gem5	Skylake-like
ISA	ARM	ARM
CPU	Out-of-Order	Out-of-Order
Private L1I Cache	8/16/32kB, 8-way, 64B line size, 2 cycle hit latency	8/16/32kB, 8-way, 64B line size, 2 cycle hit latency
Private L1D Cache	32kB, 8-way 64B line size 2 cycle hit latency	32kB, 8-way 64B line size 2 cycle hit latency
Unified L2 Cache	256kB, 8-way 64B line size 10 cycle hit 125 cycle miss	256kB, 8-way 64B line size 10-cycle hit 125 cycle miss
Branch Predictor	TAGE-SC-L	TAGE-SC-L
BTB size	4k entries	4k entries
Fetch Target Queue	5 entries	5 entries
Fetch/Decode/ Issue/Commit	8 wide	6 wide
ROB Entries	192	224
Issue Queue	64	97
Load Queue	64	72
Store Queue	64	56
Int Registers	256	180
FP Registers	512	168

Table 4.4: Processor configurations

Modified Gem5 with Decoupled, Aggresive Frontend: We extended the fetch engine of the gem5 simulator to model the aggressive frontend found in modern processors, basing the design on the original Fetch Directed Instruction Prefetching (FDIP) framework [55]. Figure 4.2 depicts the frontend microarchitecture that we implemented in the gem5 simulator. These processors employ a Fetch Target Queue (FTQ) [25, 55, 56] to enable the branch predictor and BTB to run ahead on the predicted path decoupled from the rest of the processor pipeline. Each entry in the FTQ provides the starting address and size of the dynamic basic block in bytes. A dynamic basic block is defined as the set of instructions starting at the target of a control-flow transfer instruction and ending with (and including) the next control-flow transfer instruction. Size of the basic block in FTQ helps in issuing multiple memory

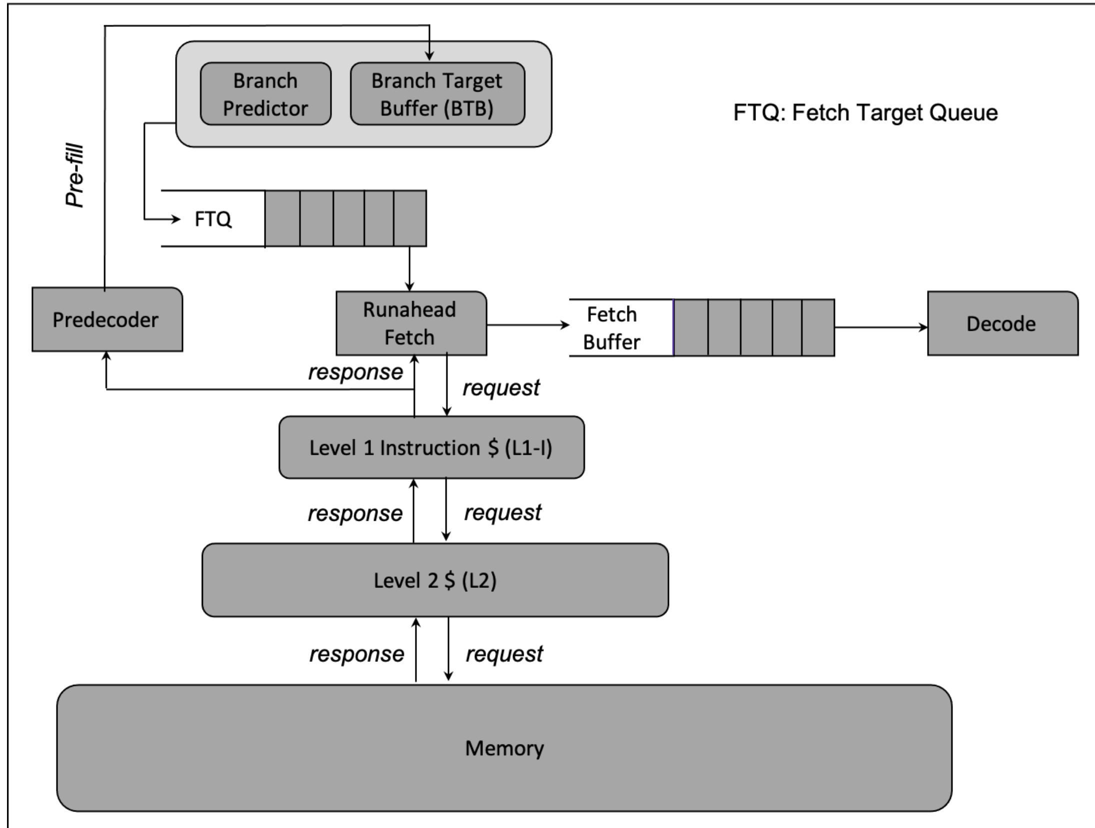


Figure 4.2: Frontend microarchitecture

requests if a basic block spans more than a single cache line. Since all our explorations with the modified gem5 included ARM ISA, wherein the size of each instruction is constant (4 bytes), this made it easier to implement aggressive run-ahead, to find the size of a dynamic basic block given its number of instructions, to find the fall-through addresess etc, without performing any pre-decoding. The hardware complexity for all such operations would increase, if an ISA with variable-length instruction encoding is used instead.

Modifications to the branch predictor and the BTB in gem5 allow it to operate at the granularity of basic blocks. Each entry in the BTB also includes the starting address of the dynamic basic block, the size of the basic block in bytes, and the type of control-flow instruction ending the basic block. The branch predictor and BTB enqueue to the FTQ up to one basic block (starting address and size) per cycle. This enqueuing stalls on BTB misses. The modeled frontend also includes a predecoder to proactively update the BTB and

minimize such enqueue stalls. Branch re-steers flush the FTQ before resuming predictions on the corrected path.

These FTQ enhancements allow the frontend to run-ahead and issue early memory requests. All experiments use an FTQ size of 5 basic blocks. This offered a balance between having sufficient starvation tolerance to hide many L1I misses (see §3.3) while keeping the front-end from becoming too aggressive in the presence of branch mispredictions. The extended run-ahead frontend necessitates having a mechanism to be able to receive the memory responses out-of-order, in the cases where earlier issued memory requests miss in L1I and later issued memory requests cause a hit in L1I. A fetch buffer was implemented where the entries get enqueued (when the respective memory requests are issued) and dequeued (when the fetch stage is ready to process the instruction bytes) in their prediction order, however, the corresponding instruction bytes are filled out-of-order when the respective memory response is received.

Modified Pipeline Diagram: Figure 4.3 depicts the pipeline diagram for few instructions from a SPEC program when run on our extended gem5 simulator in detailed CPU mode. Each line in the figure provides the details for a single dynamic instruction executed along with its disassembly. It includes the ticks at which the respective instruction passed through different pipeline stages, its address, and its dynamic instruction count in the execution. Top to bottom depicts the sequence of instructions executed in the program order. The pipeline diagram is color coded and includes a letter to distinguish the pipeline stages. 'f' indicates instruction fetch, 'd' indicates decode, 'n' indicates rename, 'p' indicates dispatch, 'i' indicates issue, 'c' indicates commit, and 'r' indicates retire. We extended the pipeline diagram to also include when the memory request was sent to L1I (indicated by 's') and when the response was received (indicated by 'v'). A green color between 's' and 'f' indicates the response was served by LRU ways in an EMISSARY L1I cache, a pink color between 's' and 'f' indicates the response was served by the high-priority ways in an

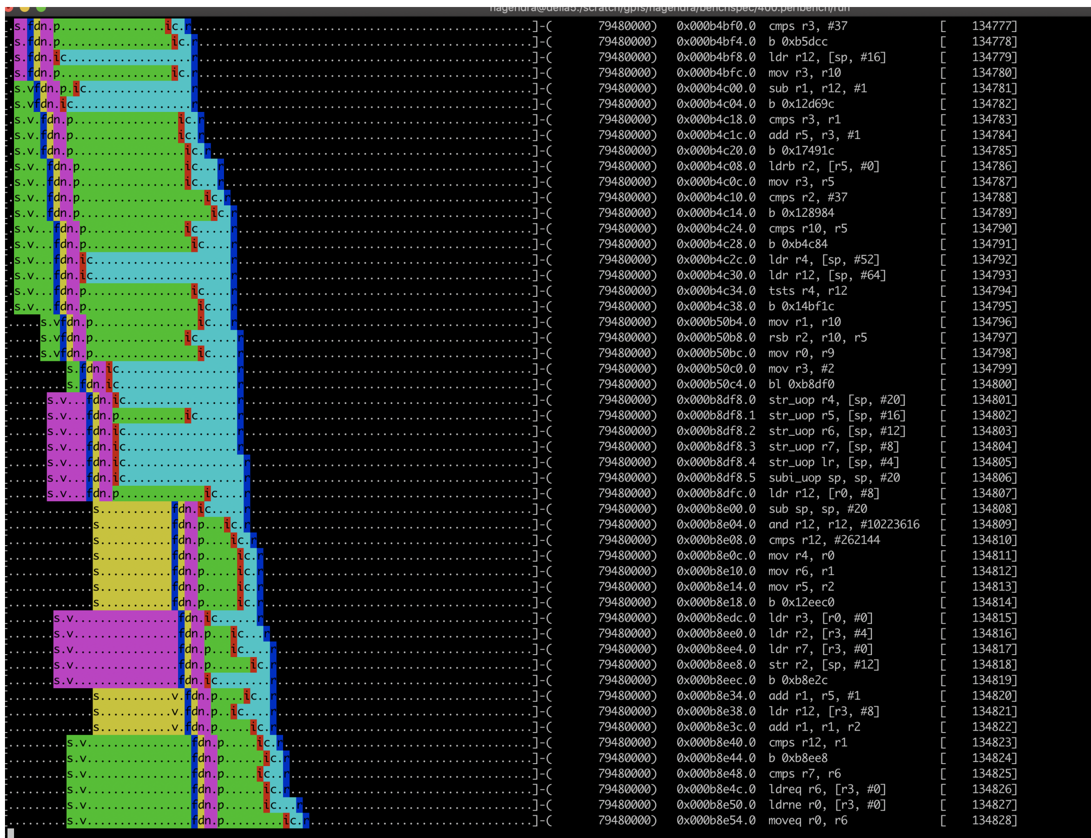


Figure 4.3: Pipeline diagram depicting out-of-order memory responses received.

EMISSARY L1I cache, and a yellow color between 's' and 'f' indicates the response was served by L2. Note many instances when the memory response for later instructions are received earlier than the issue of memory request for earlier instructions (enabled by the out-of-order filling of the fetch buffer). Such a pipeline diagram is an instance proof of the aggressive frontend implementation.

All the extensions mentioned in this section have enabled us to mimic the frontend behavior of modern architecture and perform detailed studies on them.

4.3.2 Benchmarks

L1I caches have long been under capacity pressure, and this pressure will likely grow. To model this current and future pressure, one either needs big working sets or smaller capacity caches. We chose both approaches for this study, as the latter was consistent with prior work,

making limited rough comparisons possible for the reader [35, 47, 53, 63].

We used the 22 benchmarks from SPEC CPU2006 [62], SPEC CPU2017 [61], and Tailbench [34] that could be cross-compiled and run on the simulator. Xapian from Tailbench is a web-search application. Among the 22 benchmarks, several had working set sizes too small to stress even the 8kB L1I cache. Nevertheless, we include their results for completeness. For readers interested in the set of benchmarks and configurations that stress the L1I cache, we also report geometric mean of those exceeding MPKI of 1 in addition to the geometric mean for all programs. We compiled all programs using the GCC ARM cross compiler with -O3 flags (arm-linux-gnueabihf-gcc, 9.3).

4.3.3 Policy Selection and Parameterization

§4.2 outlines a large space of possible cache replacement policies. To narrow the design space to a small and meaningful set of policies, we first select a small set of desirable policy types and then find a reasonable set of configuration parameters for these policy types with an initial exploration. The useful representative policy types chosen are M:1, M:0, M:R(r), M:S, M:S&R(r), P(N):R(r), P(N):S, and P(N):S&R(r). Ideally, we would like to find a single r and N that works well across all policies if possible. Based on prior work, we expect the best r to range from 1/2 to 1/64. [52] For an 8-way cache, useful values of N are between 1 and 6 inclusive. N value of 7 is ruled out because the lack of space for more than one low-priority line makes it subject to odd behaviors in some programs.

The search for N and r used the exploration configuration described in §4.3.2. Tables 4.5, 4.6, and 4.7 show the geometric mean speedup across all programs for each of the possible P(N) policy configurations for the 8kB, 16kB, and 32kB L1I respectively. The “#Best” row at the bottom and the “#Best” column at the right indicates the number of configurations favoring the respective column/row. We select N by selecting the row with the most wins. The clear choice here is 6 with all of the best geometric means across all the cache sizes.

For the value of r, prior work suggests 1/32 or 1/64 for M:R(r) (BIP). [52] At 8kB, as can

P(n):	S	R	R	R	R	R	R	S&R	S&R	S&R	S&R	S&R	S&R	#Best
		(1/2)	(1/4)	(1/8)	(1/16)	(1/32)	(1/64)	(1/2)	(1/4)	(1/8)	(1/16)	(1/32)	(1/64)	
N=1	0.40%	0.27%	0.38%	0.14%	0.41%	0.63%	**0.67%**	0.63%	0.91%	0.87%	1.18%	1.12%	**1.15%**	0
N=2	1.09%	0.41%	0.65%	0.91%	1.05%	1.10%	**1.21%**	1.59%	1.76%	2.05%	2.25%	2.12%	**2.38%**	0
N=3	1.67%	0.66%	1.03%	1.35%	1.60%	1.68%	**1.96%**	2.45%	2.74%	3.08%	3.27%	3.35%	**3.43%**	0
N=4	2.27%	0.80%	1.45%	2.07%	2.52%	2.70%	**2.85%**	3.28%	3.87%	4.03%	4.35%	4.35%	**4.36%**	0
N=5	3.16%	1.03%	2.23%	2.82%	3.27%	3.61%	**3.81%**	4.05%	4.92%	5.34%	5.47%	5.41%	**5.58%**	0
N=6	3.32%	1.58%	2.71%	3.53%	4.02%	4.42%	**4.56%**	4.80%	5.66%	6.12%	6.19%	6.35%	**6.43%**	**13**
#Best	-	0	0	0	0	0	**6**	0	0	0	0	0	**6**	-

Table 4.5: Geomean speedup across all configurations for various values of r and N when run on a system with 8kB L1 I-cache.

P(n):	S	R	R	R	R	R	R	S&R	S&R	S&R	S&R	S&R	S&R	#Best
		(1/2)	(1/4)	(1/8)	(1/16)	(1/32)	(1/64)	(1/2)	(1/4)	(1/8)	(1/16)	(1/32)	(1/64)	
N=1	0.29%	0.18%	0.17%	0.03%	0.08%	**0.26%**	0.23%	0.38%	0.40%	0.42%	0.36%	**0.44%**	0.27%	0
N=2	0.52%	0.36%	0.20%	0.31%	**0.55%**	0.27%	0.47%	0.66%	0.75%	0.76%	0.89%	0.73%	**0.90%**	0
N=3	0.98%	0.42%	0.53%	0.64%	0.75%	**0.78%**	0.76%	1.01%	1.25%	1.08%	1.19%	1.25%	**1.31%**	0
N=4	1.33%	0.53%	0.73%	1.09%	1.05%	1.02%	**1.11%**	1.40%	1.47%	1.64%	**1.65%**	1.58%	1.53%	0
N=5	1.46%	0.74%	1.16%	1.24%	1.36%	**1.52%**	1.36%	1.87%	1.98%	1.92%	**2.03%**	1.98%	1.87%	0
N=6	1.87%	1.22%	1.51%	1.74%	**1.77%**	1.75%	1.66%	2.06%	2.23%	2.28%	2.19%	**2.33%**	2.18%	**13**
#Best	-	0	0	0	2	**3**	1	0	0	0	**2**	**2**	**2**	-

Table 4.6: Geomean speedup across all configurations for various values of r and N when run on a system with 16kB L1 I-cache.

P(n):	S	R	R	R	R	R	R	S&R	S&R	S&R	S&R	S&R	S&R	#Best
		(1/2)	(1/4)	(1/8)	(1/16)	(1/32)	(1/64)	(1/2)	(1/4)	(1/8)	(1/16)	(1/32)	(1/64)	
N=1	0.06%	-0.11%	**-0.08%**	**-0.08%**	-0.09%	-0.19%	-0.13%	-0.03%	-0.05%	**0.09%**	0.00%	0.02%	**0.09%**	0
N=2	0.02%	**0.13%**	-0.01%	0.04%	0.04%	0.10%	-0.03%	0.06%	0.04%	**0.15%**	0.06%	0.10%	**0.15%**	0
N=3	0.17%	0.02%	0.07%	0.00%	0.01%	**0.26%**	0.05%	0.13%	0.13%	**0.21%**	0.16%	0.19%	0.17%	0
N=4	0.10%	0.02%	**0.22%**	0.08%	0.19%	0.09%	0.10%	0.24%	0.26%	**0.34%**	0.26%	0.28%	0.25%	0
N=5	0.25%	0.16%	0.15%	**0.22%**	0.13%	0.19%	0.19%	0.34%	0.35%	**0.41%**	**0.41%**	0.38%	0.30%	1
N=6	0.29%	0.22%	**0.33%**	0.26%	0.28%	0.28%	0.27%	0.40%	0.40%	0.45%	0.38%	**0.46%**	0.31%	**12**
#Best	-	1	**3**	2	0	1	0	0	0	**5**	1	1	2	-

Table 4.7: Geomean speedup across all configurations for various values of r and N when run on a system with 32kB L1 I-cache.

be seen in the Table 4.5, 1/64 performs better. At 16kB, as can be seen in the Table 4.6, 1/16, 1/32, 1/64 perform similarly. At 32kB, 1/8 performs slightly better. In light of these results and the desire to be consistent with prior work, we select 1/32 for r for all configurations with R(r). We note using the results at the other values of r for the remainder of this chapter would not change its conclusions but are omitted beyond this section.

4.3.4 Performance

Figures 4.4 - 4.21 show the speedup versus MPKI (left columns) and speedup versus change in starvation cycles for committed instructions (right columns) for all the programs. Programs in these figures are repeated at all L1I sizes (8kB, 16kB, and 32kB) to depict the change in behavior as the baseline MPKI changes. Programs 126.gcc, 176.gcc, 253.perlbmk, 400.perlbench, 445.gobmk, 458.sjeng, 464.h264ref, and Xapian have a large working set size, and hence depict certain pattern in their behavior across all the cache sizes, which will be covered in detail next. Remaining programs in figures 4.4 - 4.21 have a smaller working set size, and hence their MPKI is less than 0.5 for each of those in cache sizes larger than 8kB. Few other programs that had MPKI less than 0.5 even on a 8kB L1I are omitted in these figures.

Figures 4.4 - 4.21 show all values of N, from 0 to 6. For all P(N) configurations, an N of 0 is equivalent to LRU, the baseline. Lines connect P(n) to P(n+1) for each N from 0 to 6. As the maximum number of high-priority lines protected (N) increases from 1 to 6, performance tends to increase. This trend holds, in general, across all cache sizes and configurations when the baseline MPKI is greater than 1.0.

Many cases in Figures 4.4 - 4.21 clearly demonstrate that the correlation between performance and MPKI is not as strong as performance and starvation of committed instructions. For 126.gcc and 176.gcc at all the cache sizes, 445.gobmk and 458.sjeng at 8kB and 32kB, 464.h264ref at 32kB, and Xapian at 16kB and 32kB the speedup vs. MPKI graphs exhibit both increases and decreases in MPKI as performance increases. Meanwhile, the speedup

vs. change in starvation graphs for these benchmarks show a clear increase in performance as the starvations are reduced.

In cases where MPKI is less than 1.0 (for example 197.parser, 456.hmmer at all cache sizes, Xapian at 32kB etc.), the relationships among speedup, MPKI, and change in starvation are less clear. The lines formed by increasing N lack a smooth shape. At these low miss rates, the impact any cache replacement policy can have is small, and EMISSARY is no different. With few conflict misses, other effects may dominate. For a program dominated by compulsory misses, an LRU cache will quickly use all ways in the set. Since EMISSARY reserves N ways for high-priority lines, compulsory misses may compete for fewer ways in the set. In the few cases in which it does lose to LRU, it does not lose by much. Still, to address this problem, we have suggested some mechanisms which are described in detail in §4.4.

The results clearly show that it is quite possible to achieve higher performance with increased MPKI. This is the central observation of all cost-aware cache replacement policy proposals, and this observation is confirmed for the EMISSARY techniques, despite their simplicity. Not all cache misses in the modern out-of-order processors have the same cost. A significant portion of the misses are capable of tolerating latency from lower-level caches without degrading processor performance. Similarly, a significant portion of the addresses that are latency-sensitive and cause decode starvations do so every time they are accessed. EMISSARY handles both of these categories very efficiently. It assigns higher priority to starvation-prone addresses, keeping them in the cache longer even if they are not accessed frequently. EMISSARY gives lower priority to latency-tolerant addresses, but it does cache them long enough to capture as much of their locality.

Figures 4.4 - 4.21 show that all of the persistent bimodal techniques (those with P(N)) outperform LRU and their MRU insertion policy counterparts (those with M) across all programs and cache sizes. EMISSARY techniques perform well as the cache size is reduced, but the same is not true with the MRU insertion policies, which may significantly

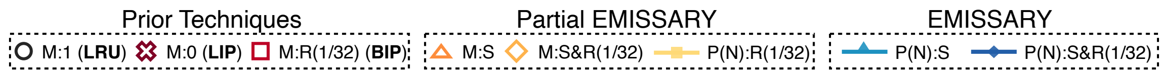
underperform the baseline LRU, as seen in 126.gcc, 132.ijpeg, 176.gcc, 253.perlbmk, 400.perlbench, 445.gobmk, 458.sjeng, and Xapian at 8kB cache. As described in §4.6, significantly underperforming LRU is a problem seen in prior cost-aware cache replacement policies, often requiring the addition of complex hardware to address.

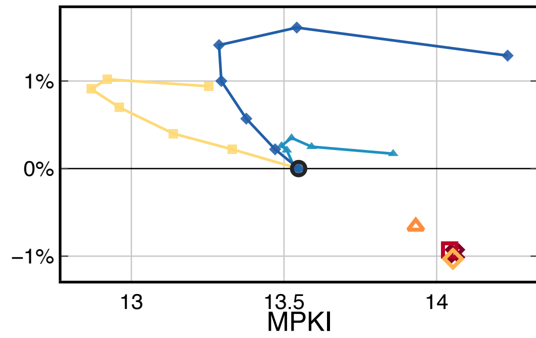
Geomtric Mean when N=6 and R=1/32

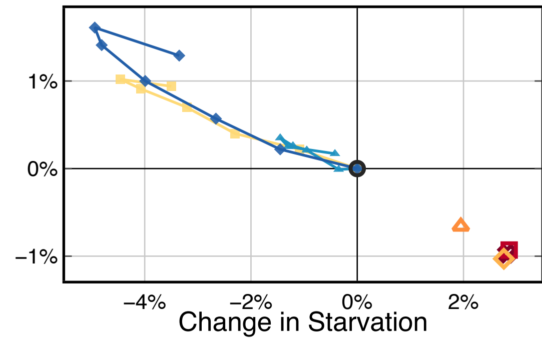
Fig. 4.22 shows the speedup and energy reduction of EMISSARY, partial EMISSARY, and other related prior work techniques for the selected configurations across all the programs and cache sizes measured. Two geometric means are shown at the right as the last set of bars. The first geometric mean is of all programs. The second geometric mean is of all programs with an MPKI greater than 1. Fig. 4.23 shows the speedup and energy reduction with the Default "gem5" model, and detailed simulations of the first 100 million instructions on a subset of programs.

Figures 4.22 and 4.23 show that EMISSARY techniques outperform all of the others in terms of speedup and energy savings, irrespective of the program region simulated. The preferred configuration, P(6):S&R(1/32), performs consistently better than P(6):S. This is because the random filter tends to require lines to prove themselves with multiple starvations before being marked high-priority. This filters single reference lines very effectively, but it also filters single decode starvation lines just as well. This is important because high-priority reservations should be allocated to lines that starve with high probability and frequency, especially since an early single starvation is possible due to branch mispredictions and warmup as new regions of code are executed.

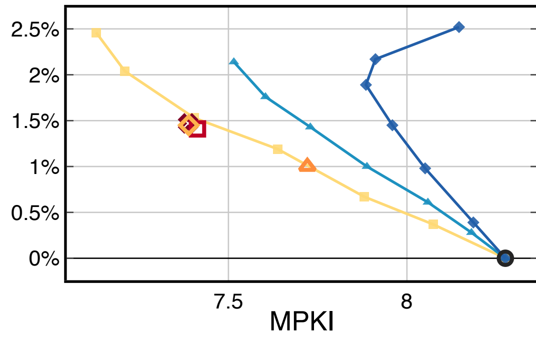
Among the MRU Insert techniques, LIP (M:0) and BIP (M:R(1/32)) generally outperform M:S and come very close to M:S&R(1/32). However, for the smaller cache sizes, the MRU Insert techniques degrade performance. This highlights that neither starvation by itself nor persistence by itself is sufficient to achieving EMISSARY's full potential. As outlined earlier, the best results in terms of overall performance and performance stability occur

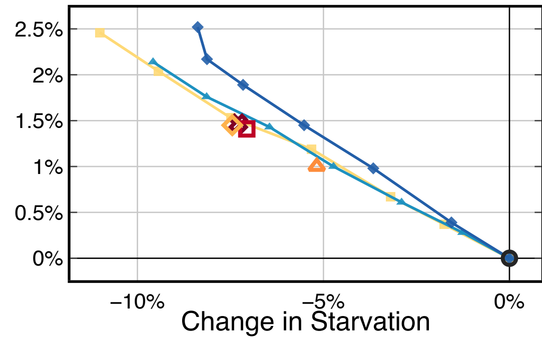




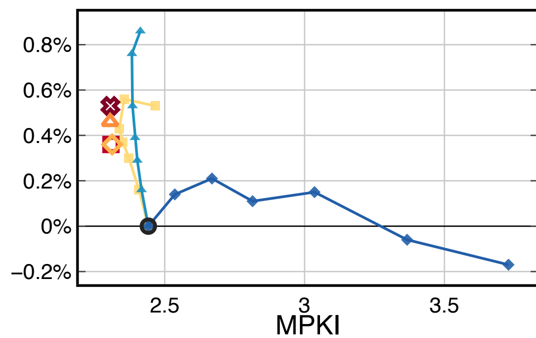


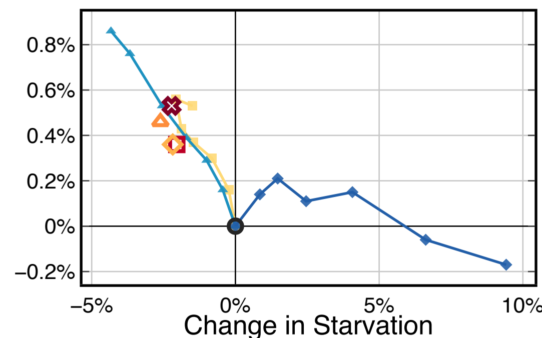
(a) 126.gcc@8kB





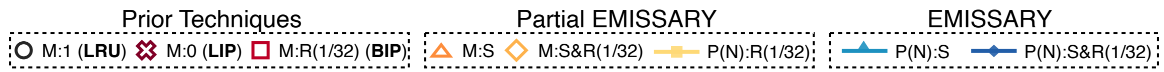
(b) 126.gcc@16kB

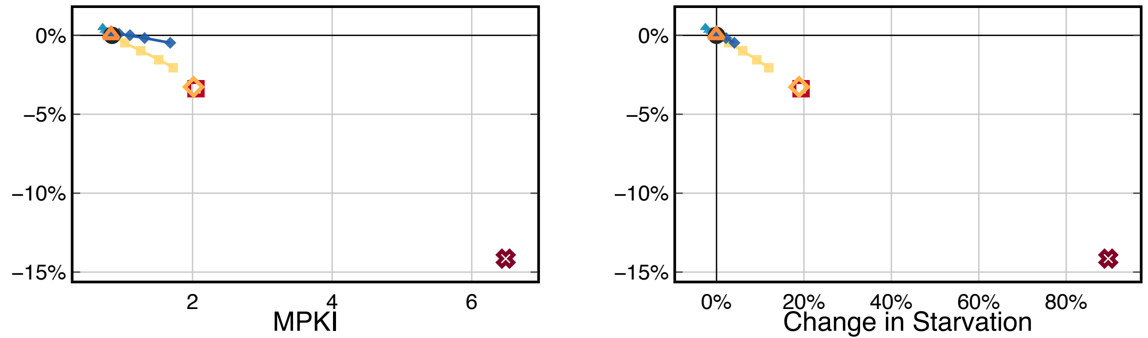




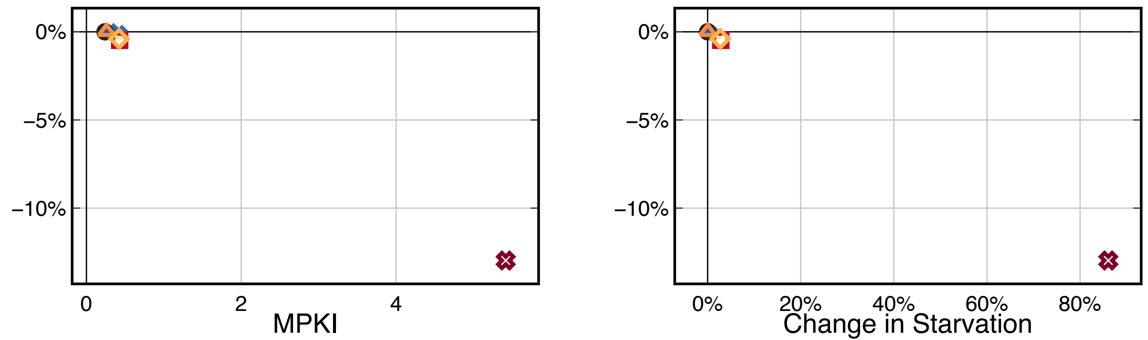
(c) 126.gcc@32kB

Figure 4.4: Speedup vs. MPKI and Speedup vs. Change in Starvation for 126.gcc

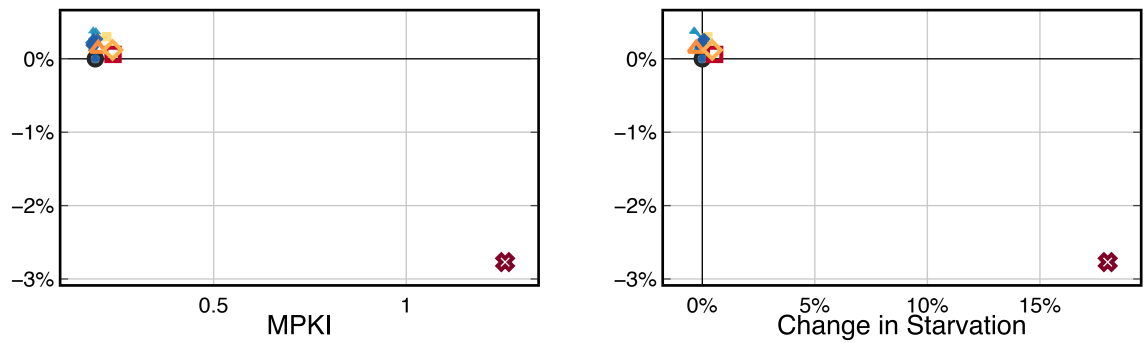




(a) 132.ijpeg@8kB

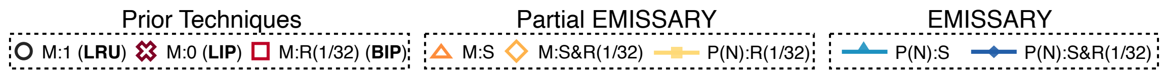


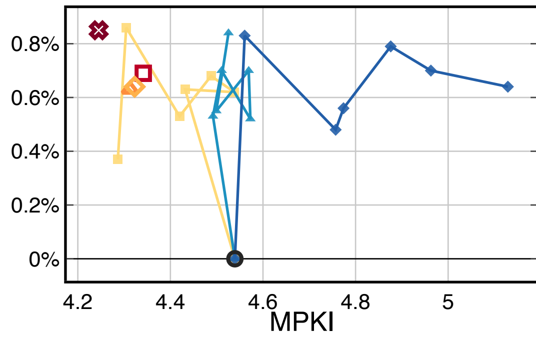
(b) 132.ijpeg@16kB

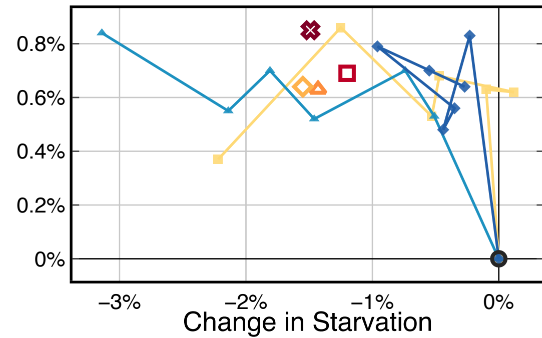


(c) 132.ijpeg@32kB

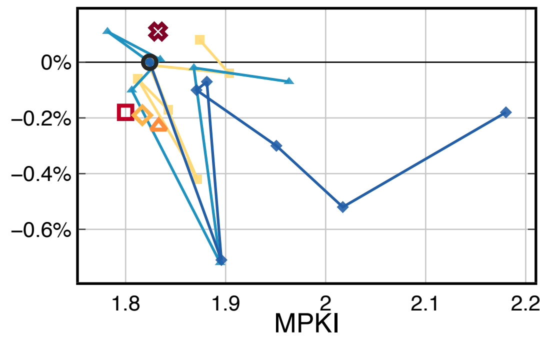
Figure 4.5: Speedup vs. MPKI and Speedup vs. Change in Starvation for 132.ijpeg

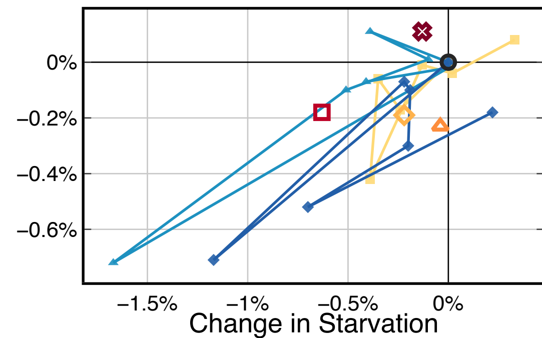




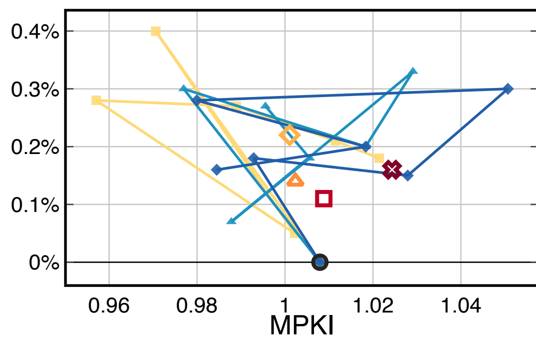


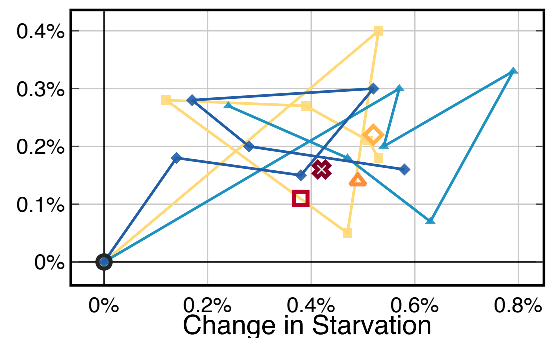
(a) 171.swim@8kB





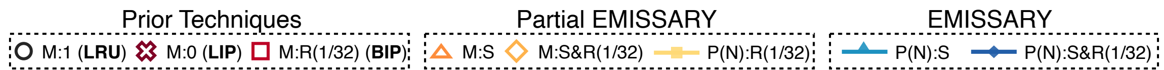
(b) 171.swim@16kB

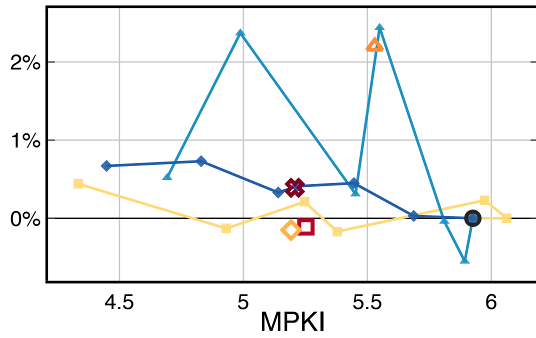


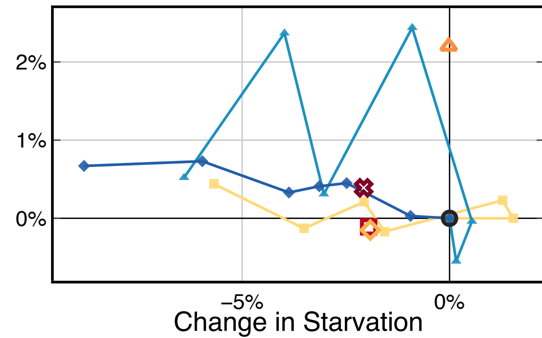


(c) 171.swim@32kB

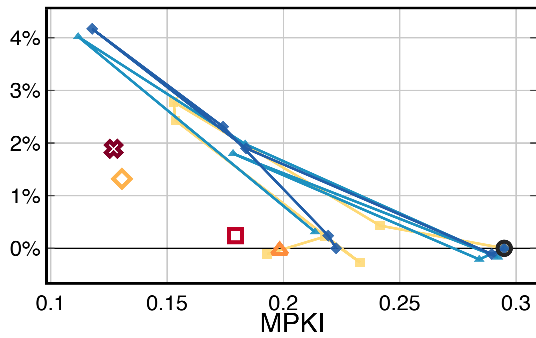
Figure 4.6: Speedup vs. MPKI and Speedup vs. Change in Starvation for 171.swim

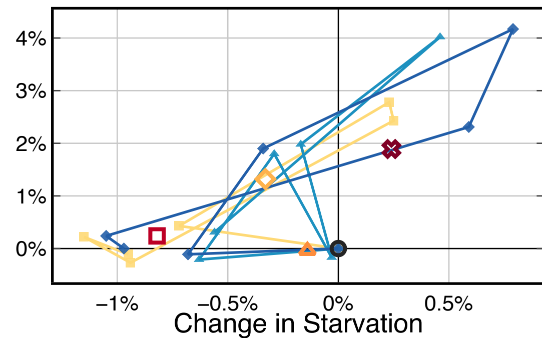




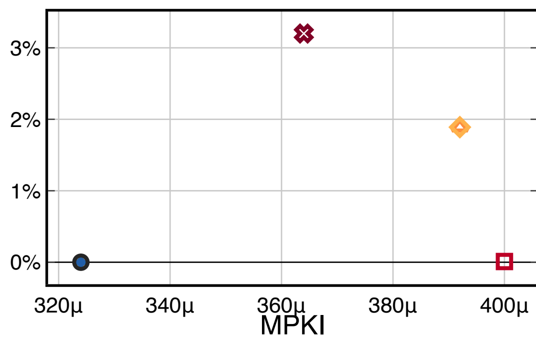


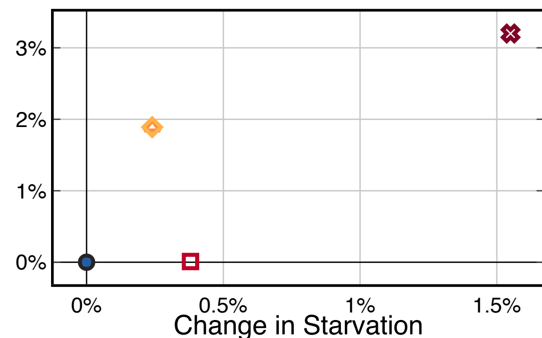
(a) 175.vpr@8kB





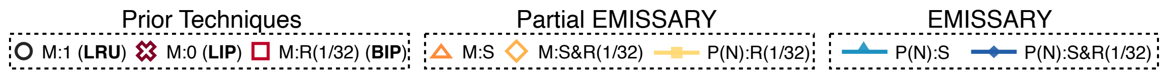
(b) 175.vpr@16kB

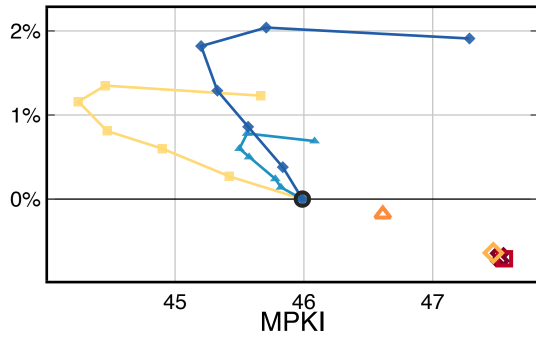


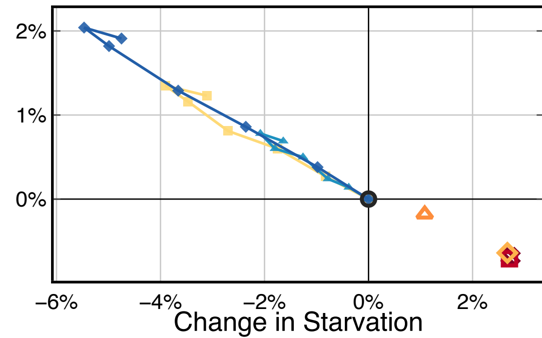


(c) 175.vpr@32kB

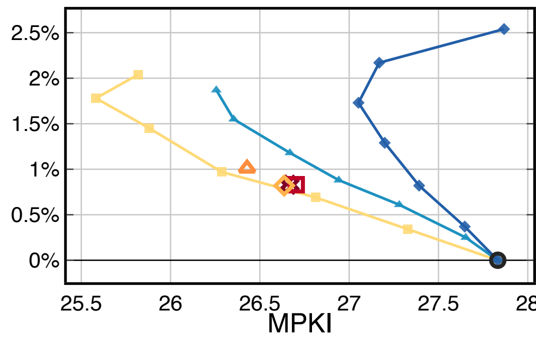
Figure 4.7: Speedup vs. MPKI and Speedup vs. Change in Starvation for 175.vpr

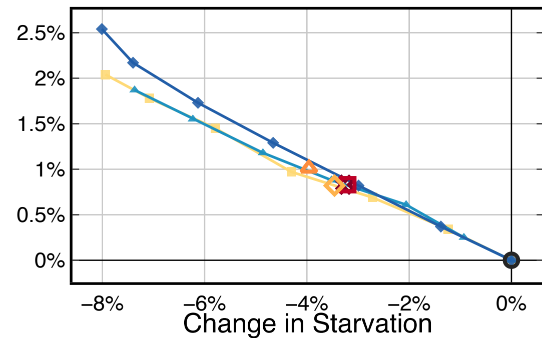




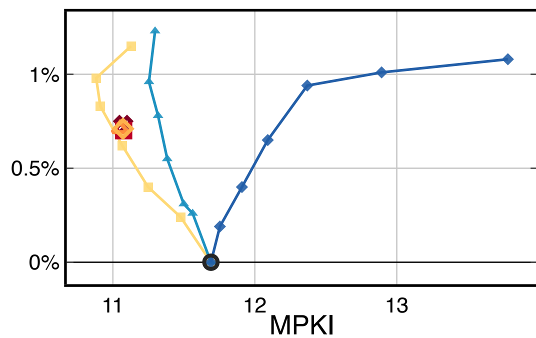


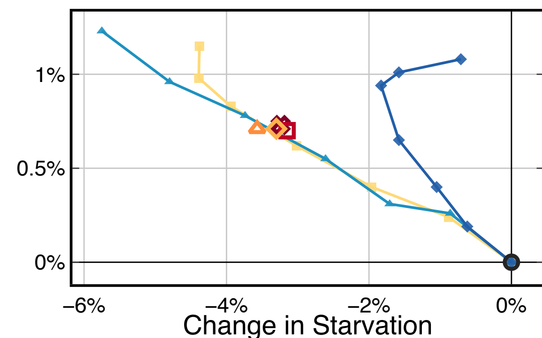
(a) 176.gcc@8kB





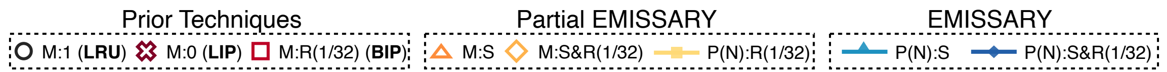
(b) 176.gcc@16kB

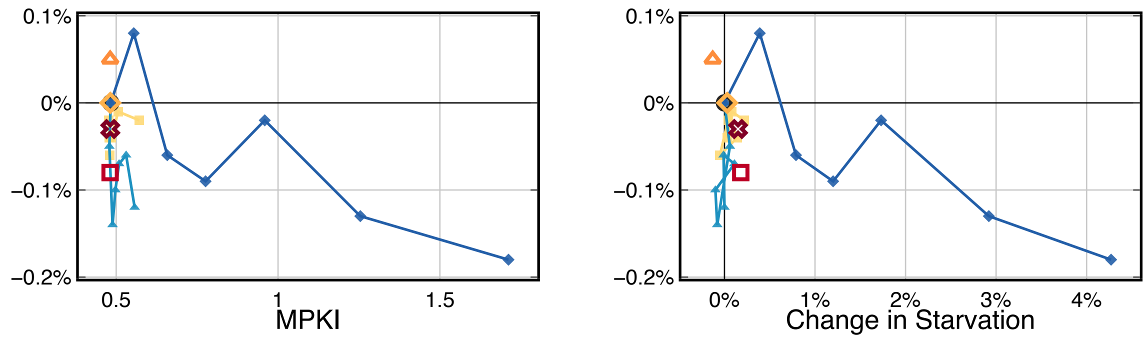




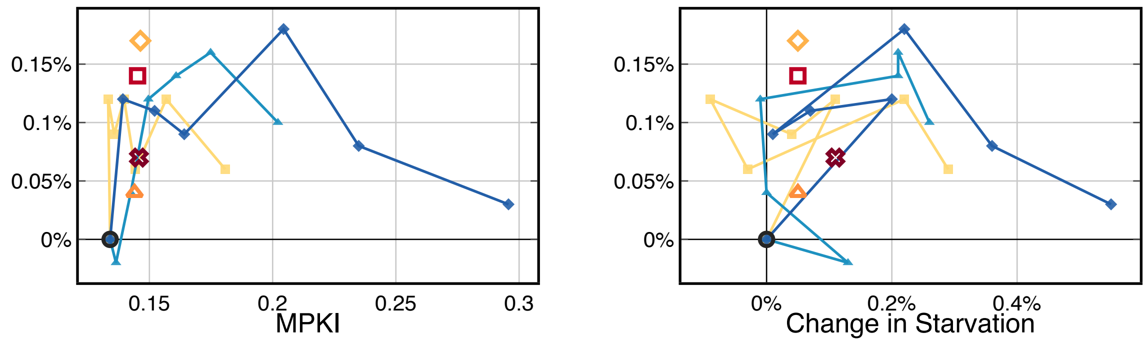
(c) 176.gcc@32kB

Figure 4.8: Speedup vs. MPKI and Speedup vs. Change in Starvation for 176.gcc

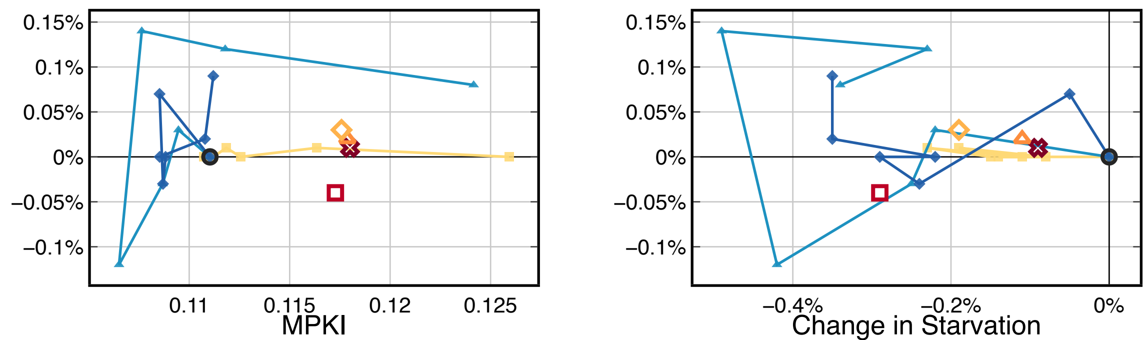




(a) 197.parser@8kB



(b) 197.parser@16kB



(c) 197.parser@32kB

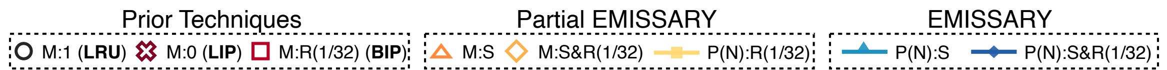
Figure 4.9: Speedup vs. MPKI and Speedup vs. Change in Starvation for 197.parser

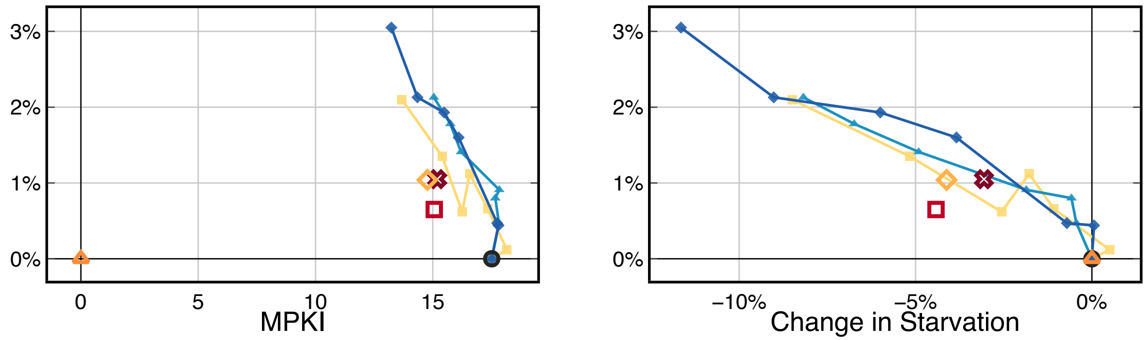
(a) 253.perlbmk@8kB

(b) 253.perlbmk@16kB

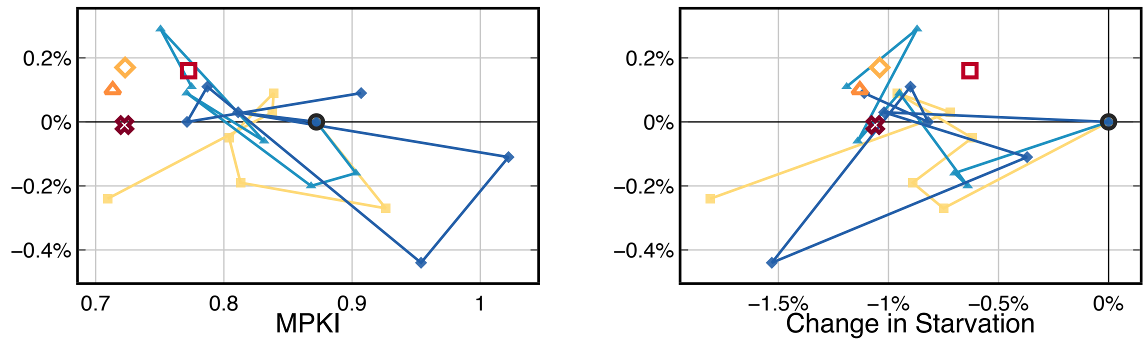
(c) 253.perlbmk@32kB

Figure 4.10: Speedup vs. MPKI and Speedup vs. Change in Starvation for 253.perlbmk

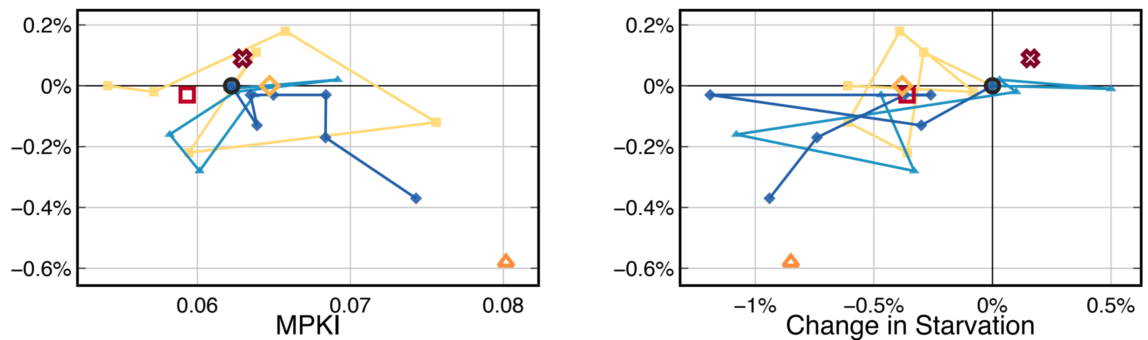




(a) 254.gap@8kB

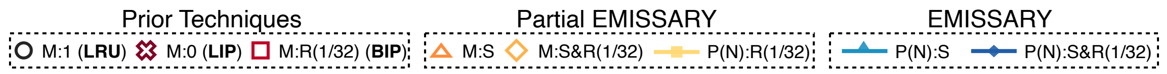


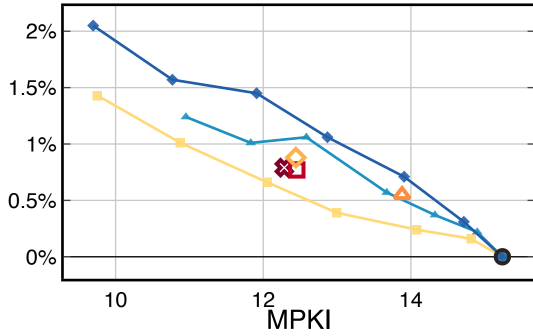
(b) 254.gap@16kB

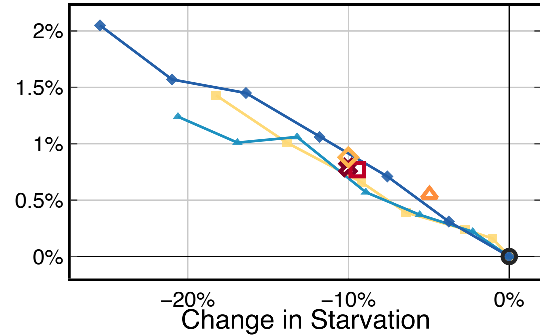


(c) 254.gap@32kB

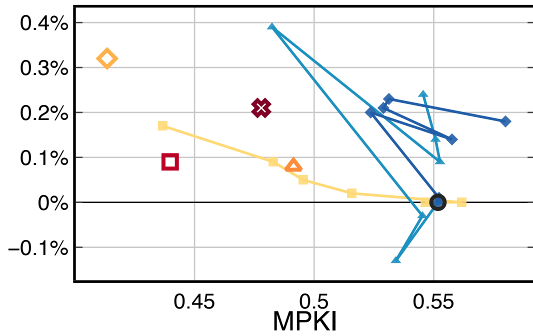
Figure 4.11: Speedup vs. MPKI and Speedup vs. Change in Starvation for 254.gap

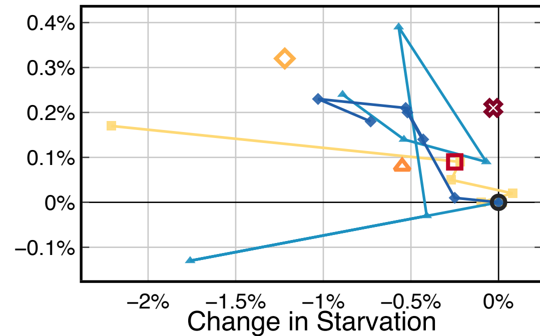




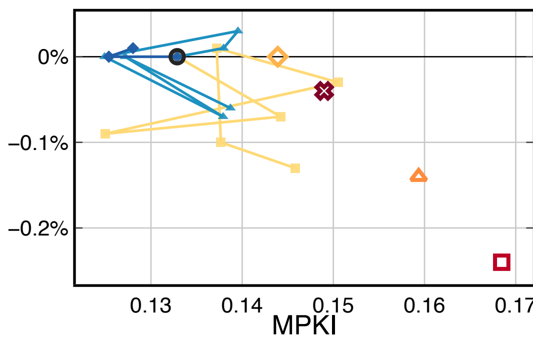


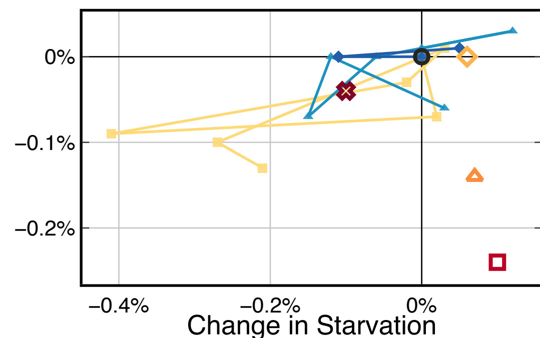
(a) 300.twolf@8kB





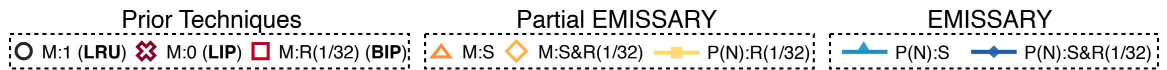
(b) 300.twolf@16kB

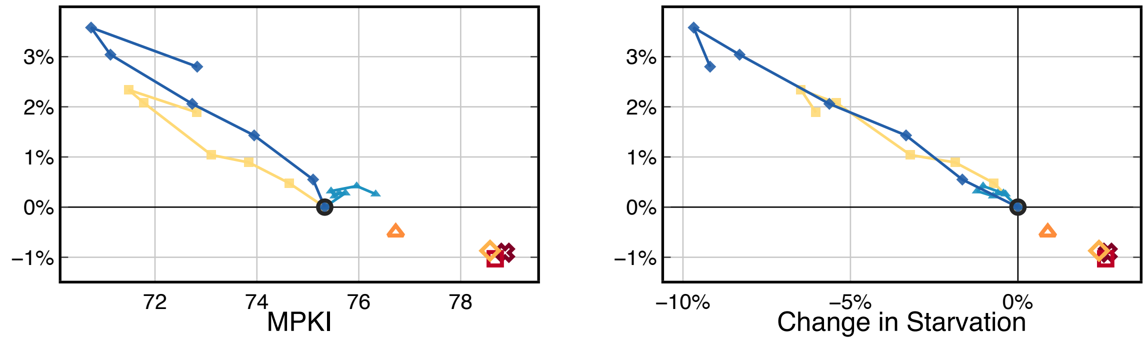




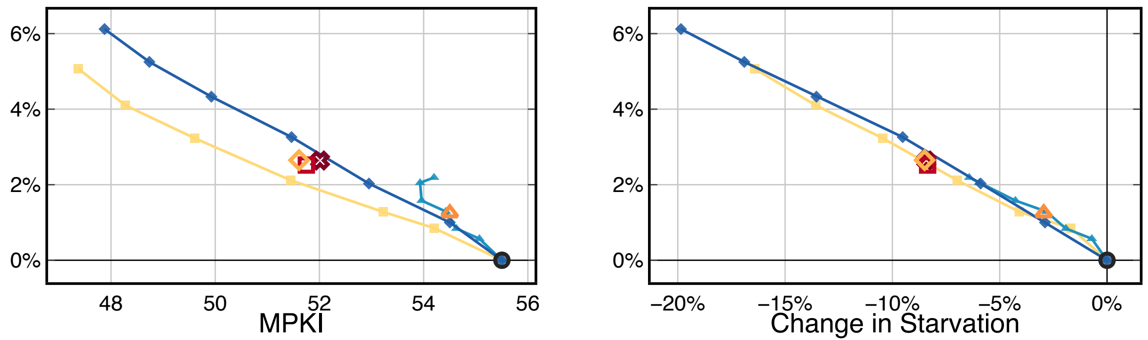
(c) 300.twolf@32kB

Figure 4.12: Speedup vs. MPKI and Speedup vs. Change in Starvation for 300.twolf

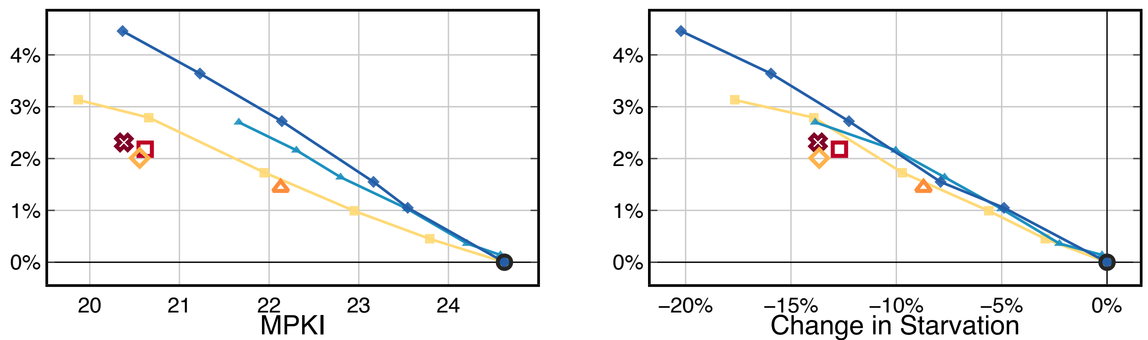




(a) 400.perlbench@8kB

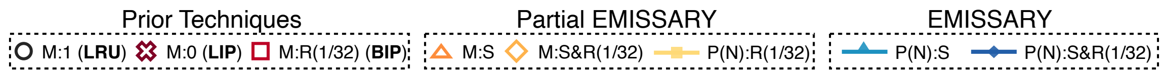


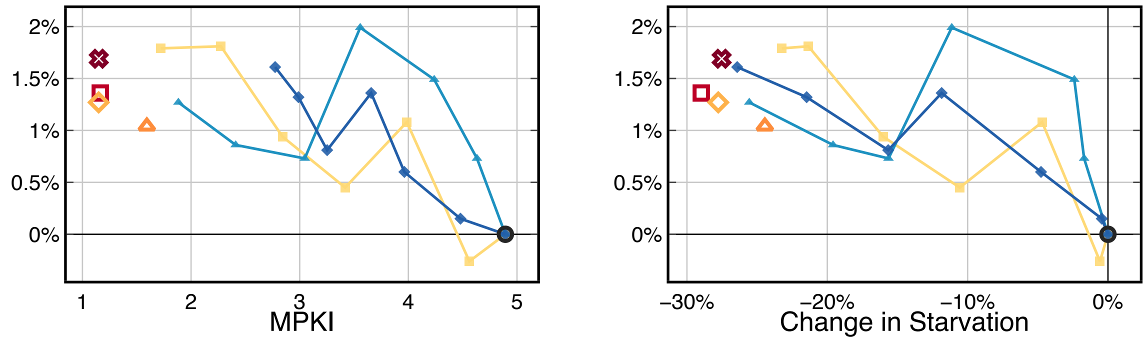
(b) 400.perlbench@16kB



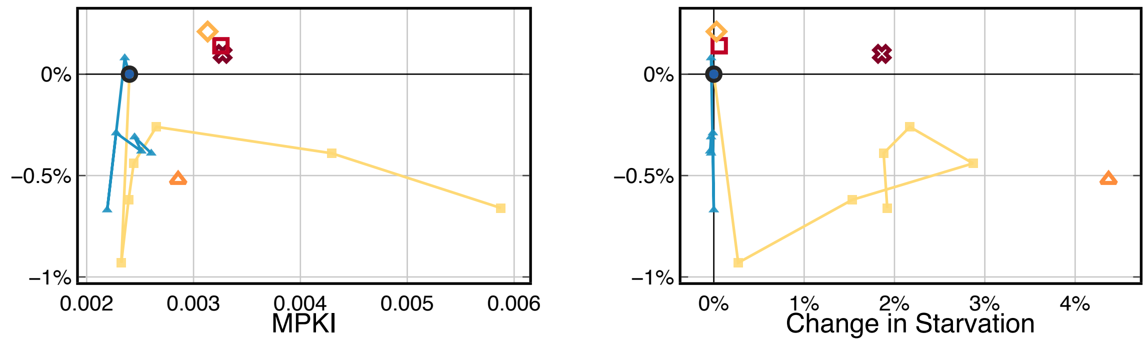
(c) 400.perlbench@32kB

Figure 4.13: Speedup vs. MPKI and Speedup vs. Change in Starvation for 400.perlbench

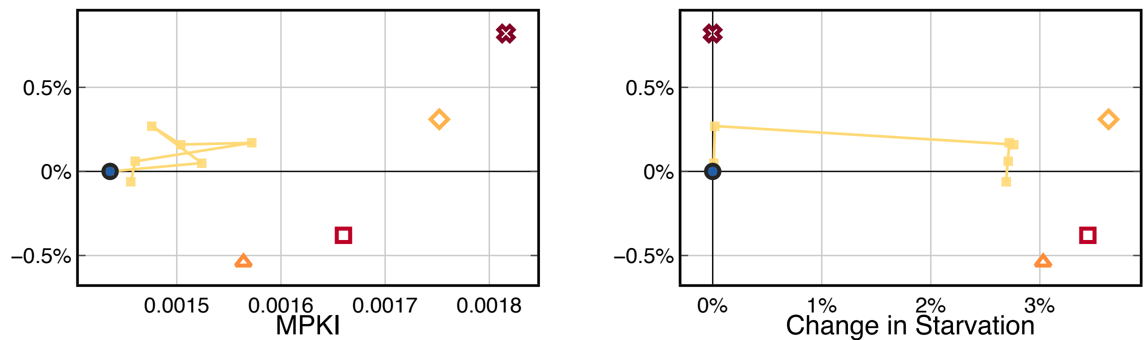




(a) 401.bzip2@8kB

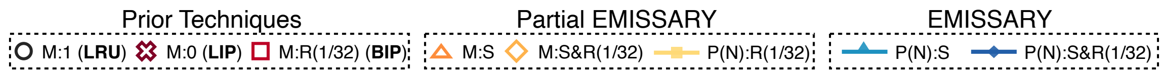


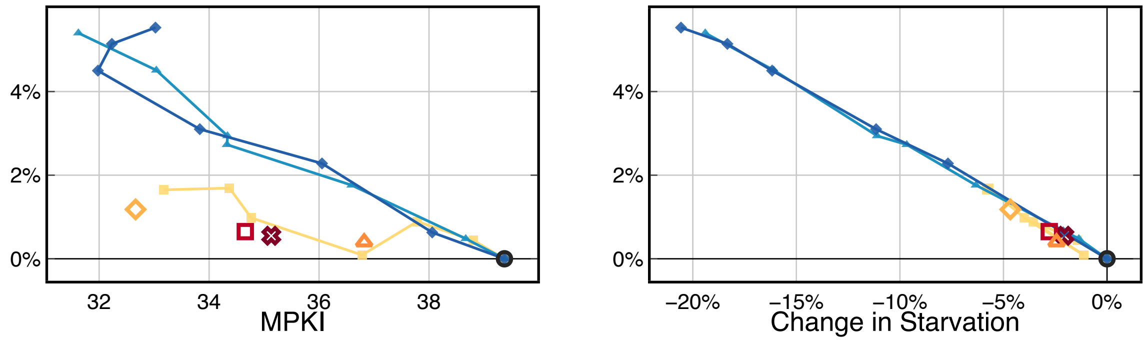
(b) 401.bzip2@16kB



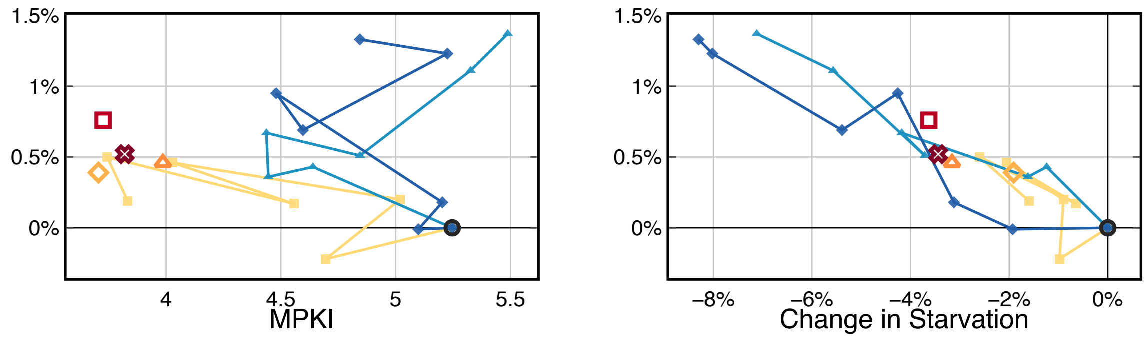
(c) 401.bzip2@32kB

Figure 4.14: Speedup vs. MPKI and Speedup vs. Change in Starvation for 401.bzip2

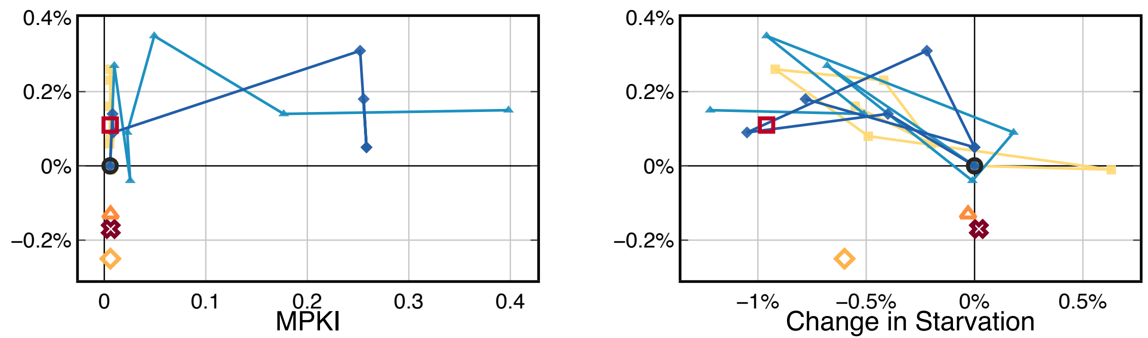




(a) 444.namd@8kB

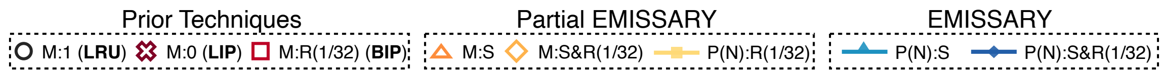


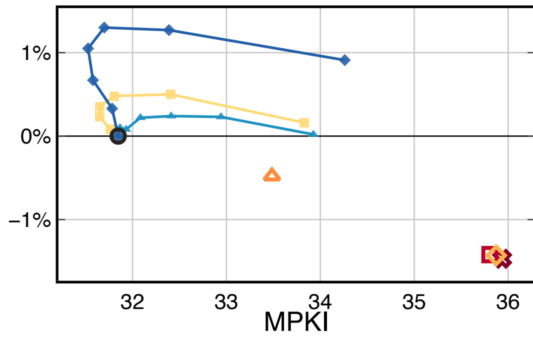
(b) 444.namd@16kB

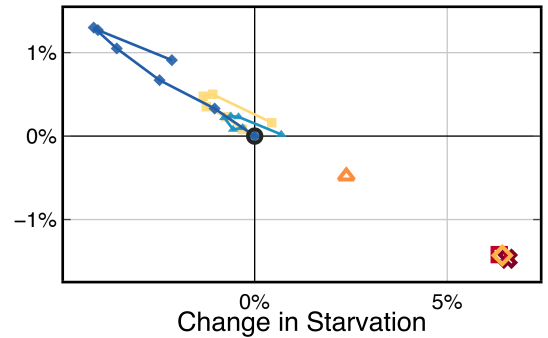


(c) 444.namd@32kB

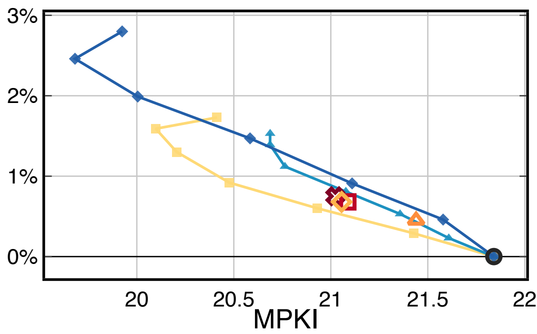
Figure 4.15: Speedup vs. MPKI and Speedup vs. Change in Starvation for 444.namd

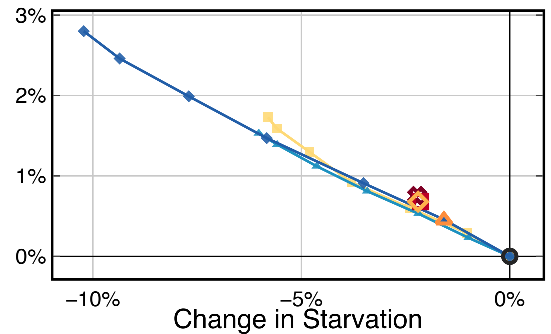




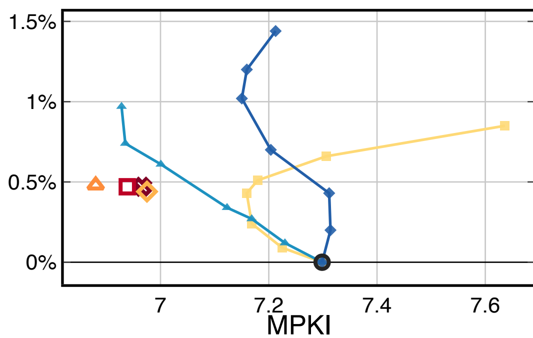


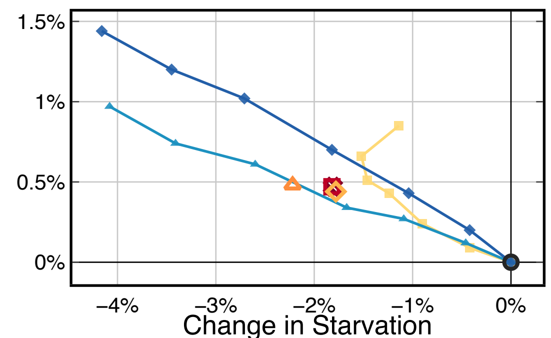
(a) 445.gobmk@8kB





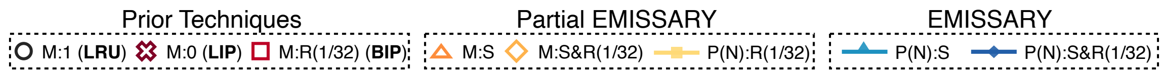
(b) 445.gobmk@16kB

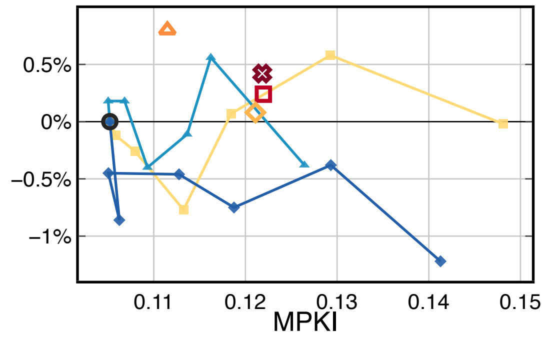


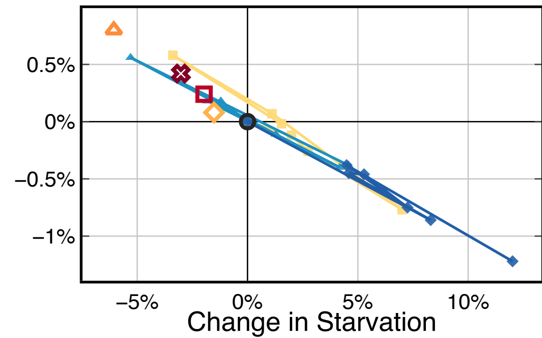


(c) 445.gobmk@32kB

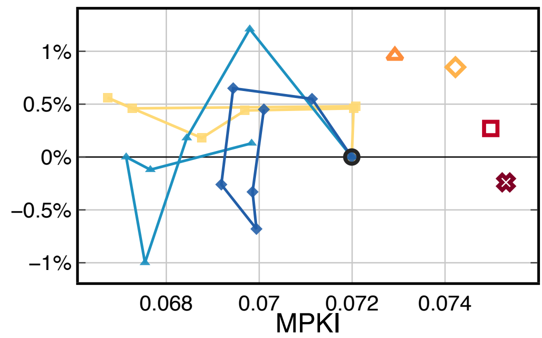
Figure 4.16: Speedup vs. MPKI and Speedup vs. Change in Starvation for 445.gobmk

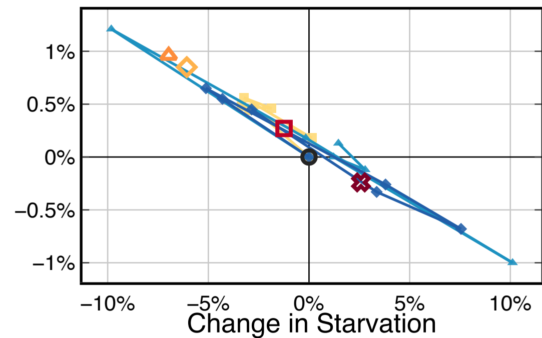




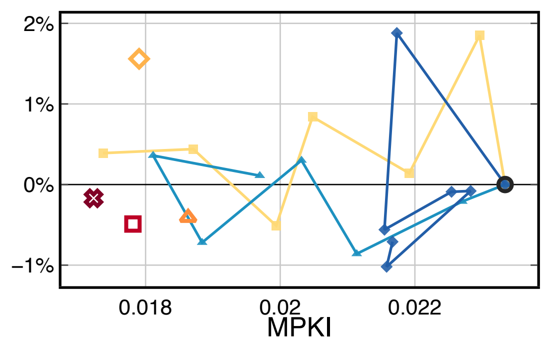


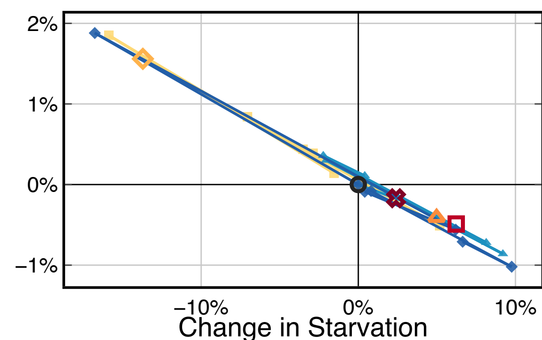
(a) 456.hmmer@8kB





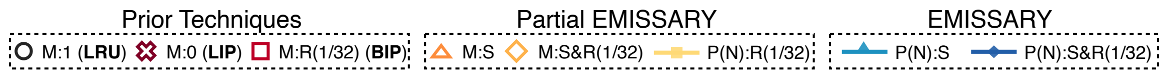
(b) 456.hmmer@16kB

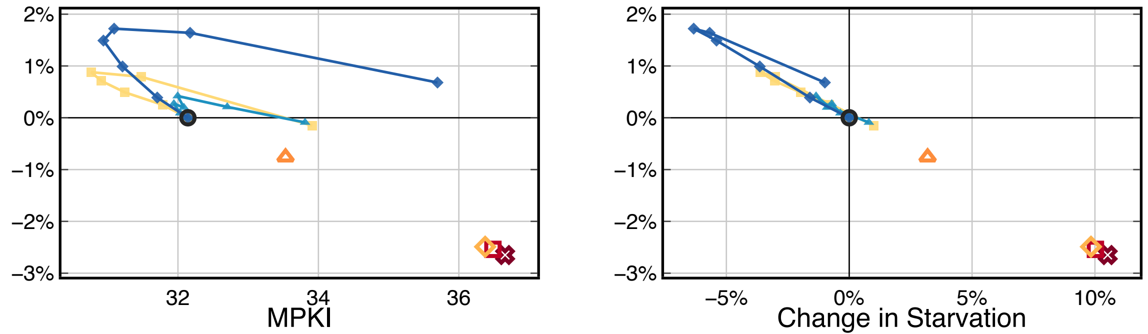




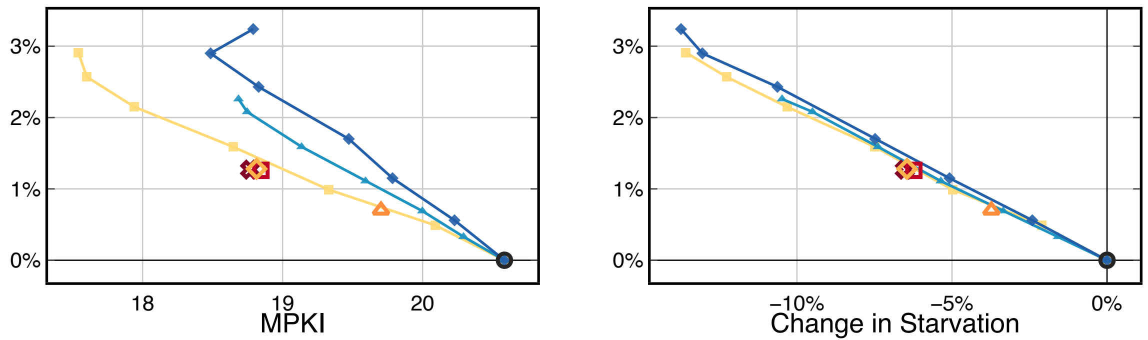
(c) 456.hmmer@32kB

Figure 4.17: Speedup vs. MPKI and Speedup vs. Change in Starvation for 456.hmmer

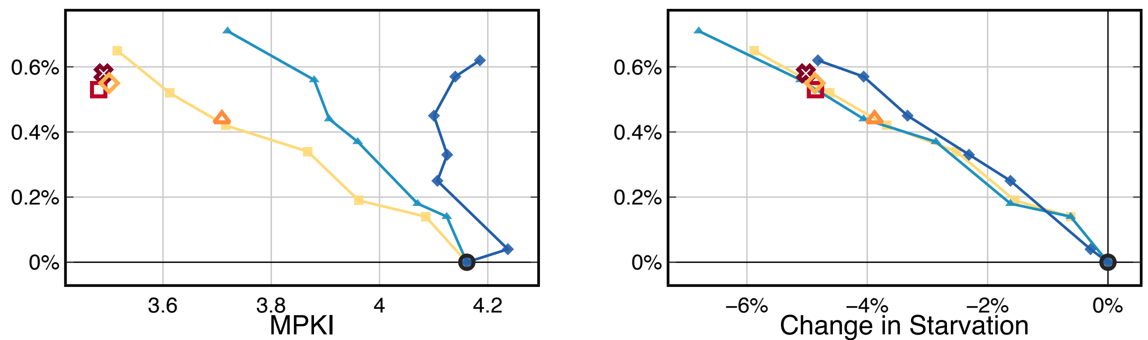




(a) 458.sjeng@8kB

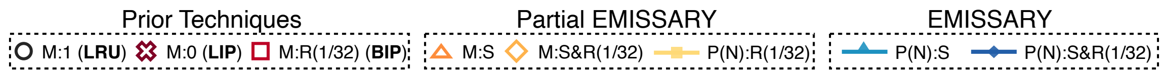


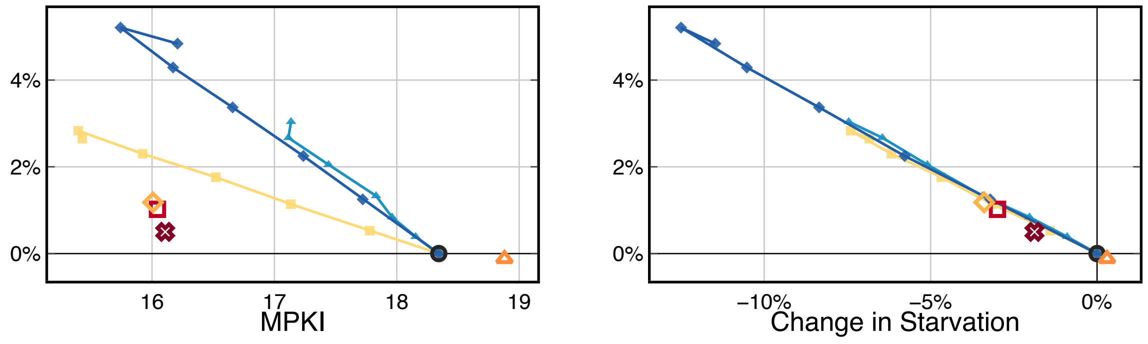
(b) 458.sjeng@16kB



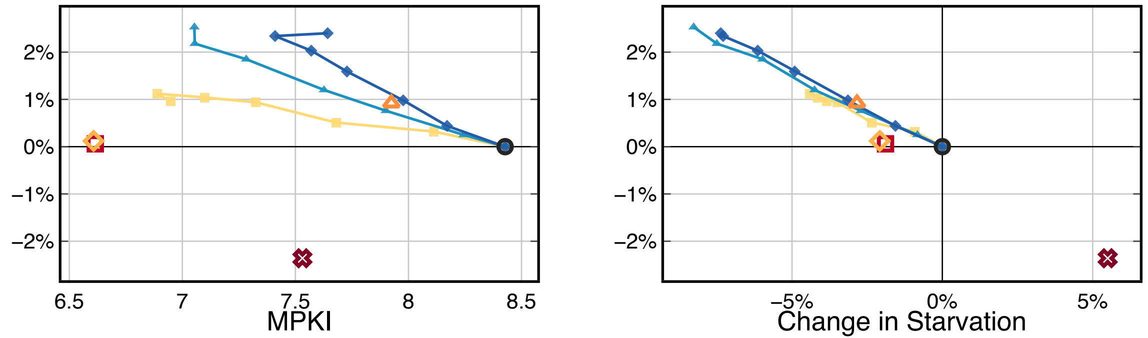
(c) 458.sjeng@32kB

Figure 4.18: Speedup vs. MPKI and Speedup vs. Change in Starvation for 458.sjeng

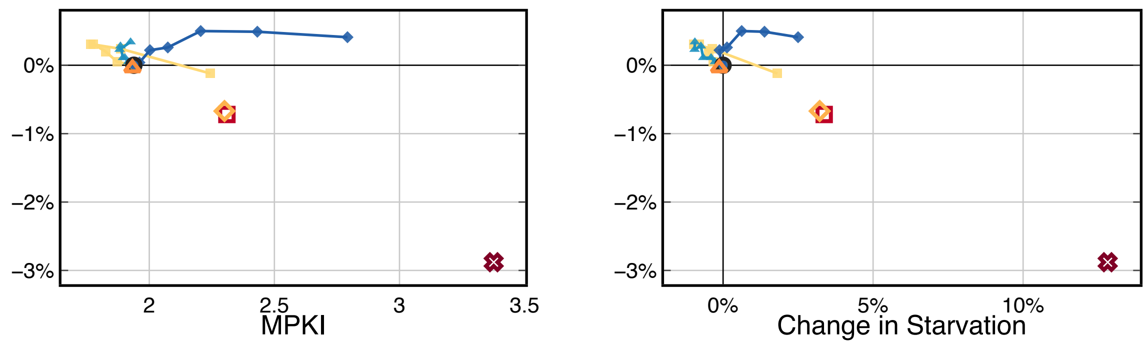




(a) 464.h264ref@8kB

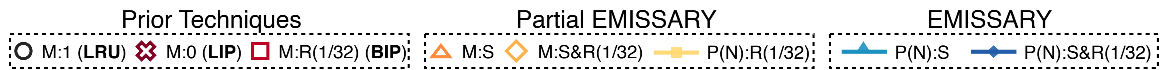


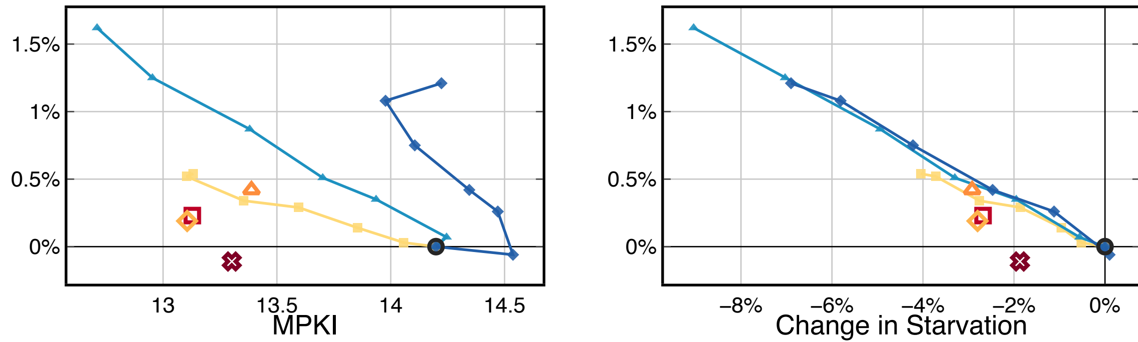
(b) 464.h264ref@16kB



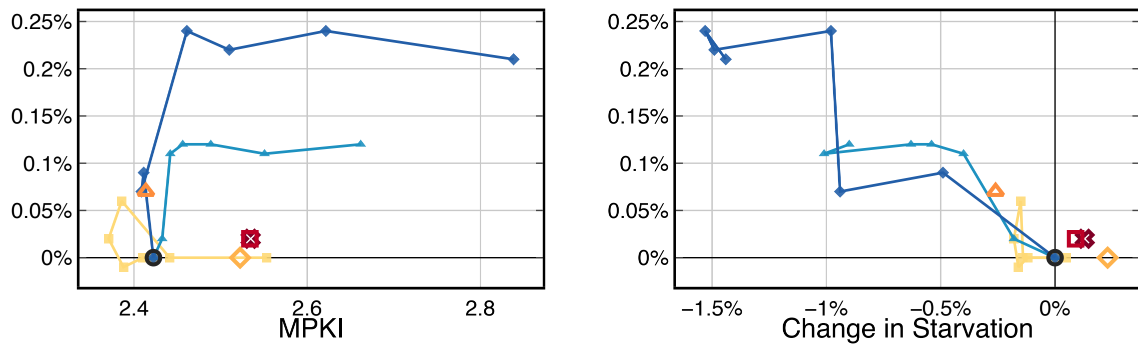
(c) 464.h264ref@32kB

Figure 4.19: Speedup vs. MPKI and Speedup vs. Change in Starvation for 464.h264ref

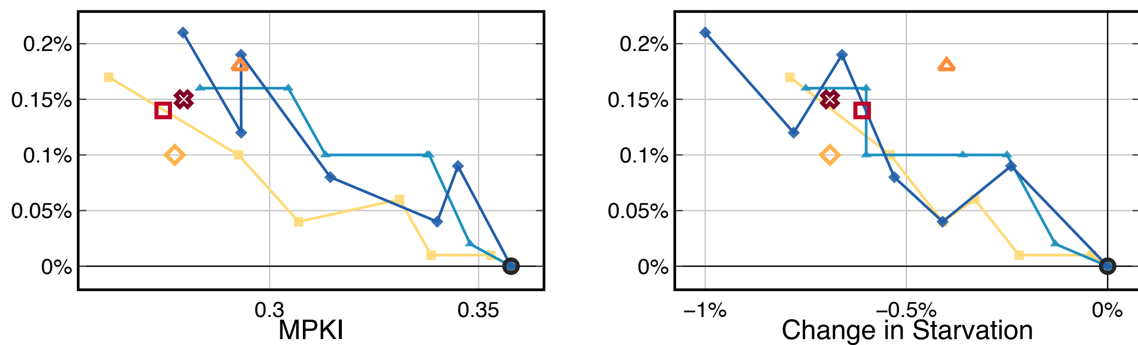




(a) 541.leela@8kB

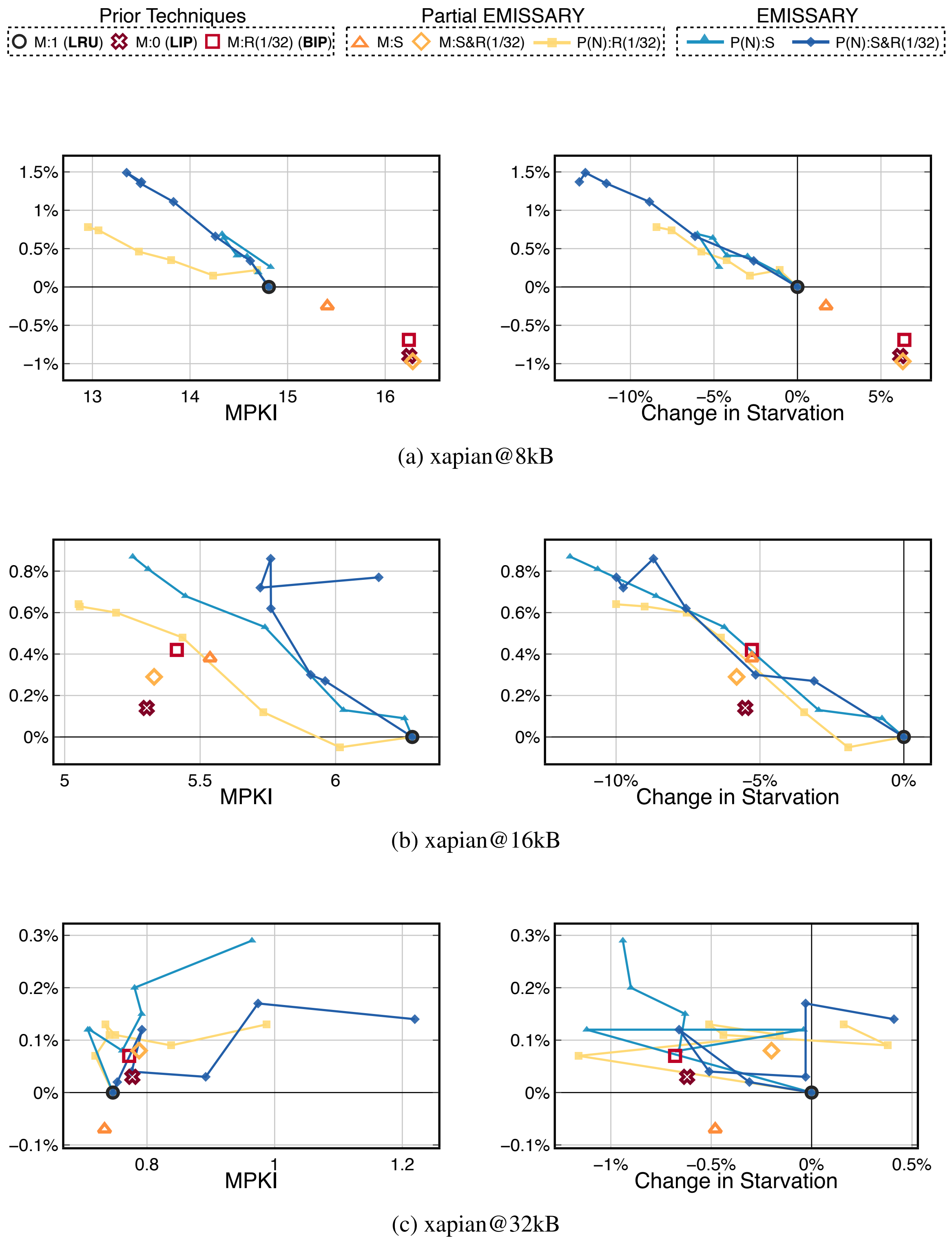


(b) 541.leela@16kB



(c) 541.leela@32kB

Figure 4.20: Speedup vs. MPKI and Speedup vs. Change in Starvation for 541.leela



(a) xapian@8kB

(b) xapian@16kB

(c) xapian@32kB

Figure 4.21: Speedup vs. MPKI and Speedup vs. Change in Starvation for Xapian (Tailbench)

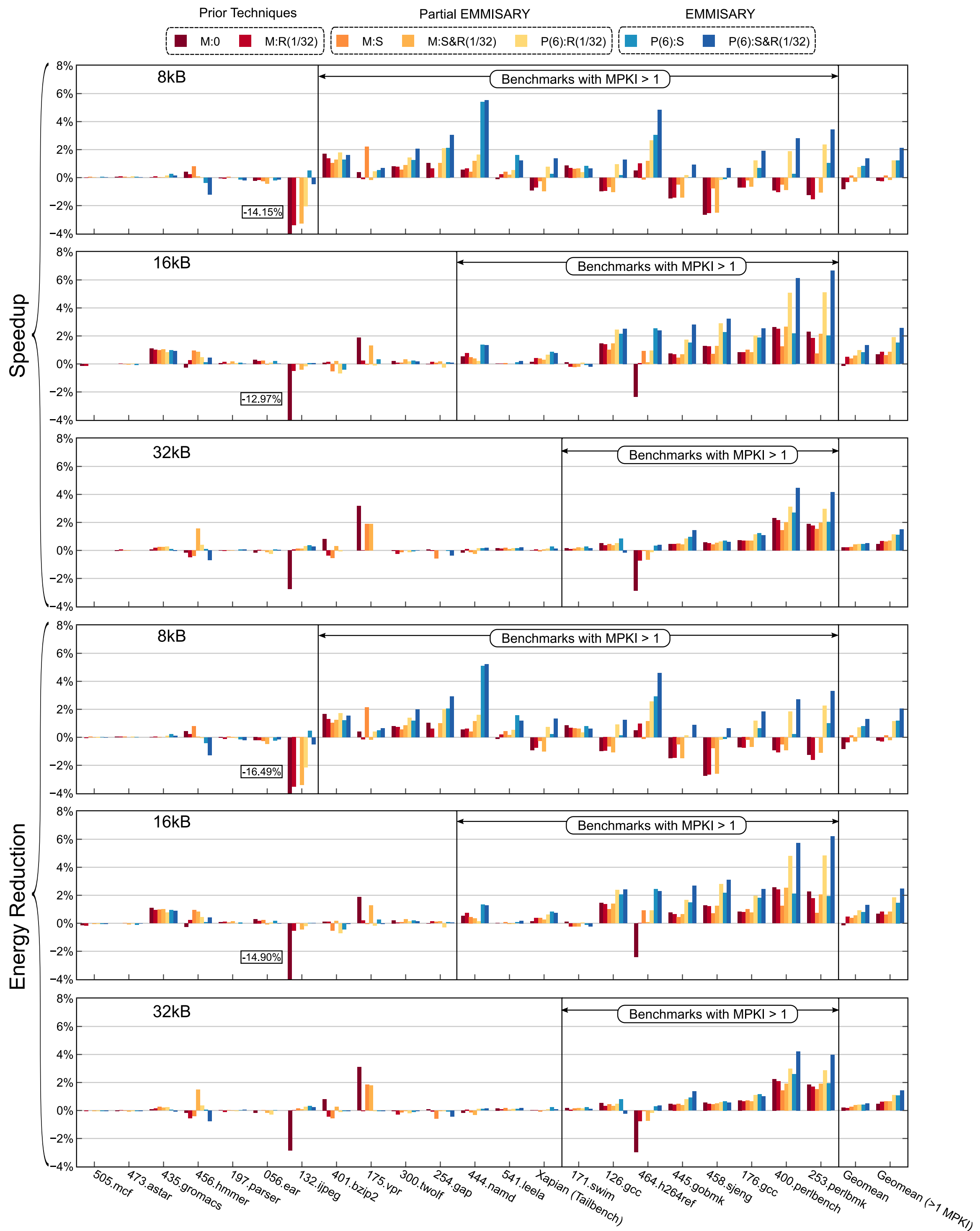


Figure 4.22: Speedup and Energy Reduction of a range of techniques relative to LRU. The programs are generally sorted by their baseline MPKI.

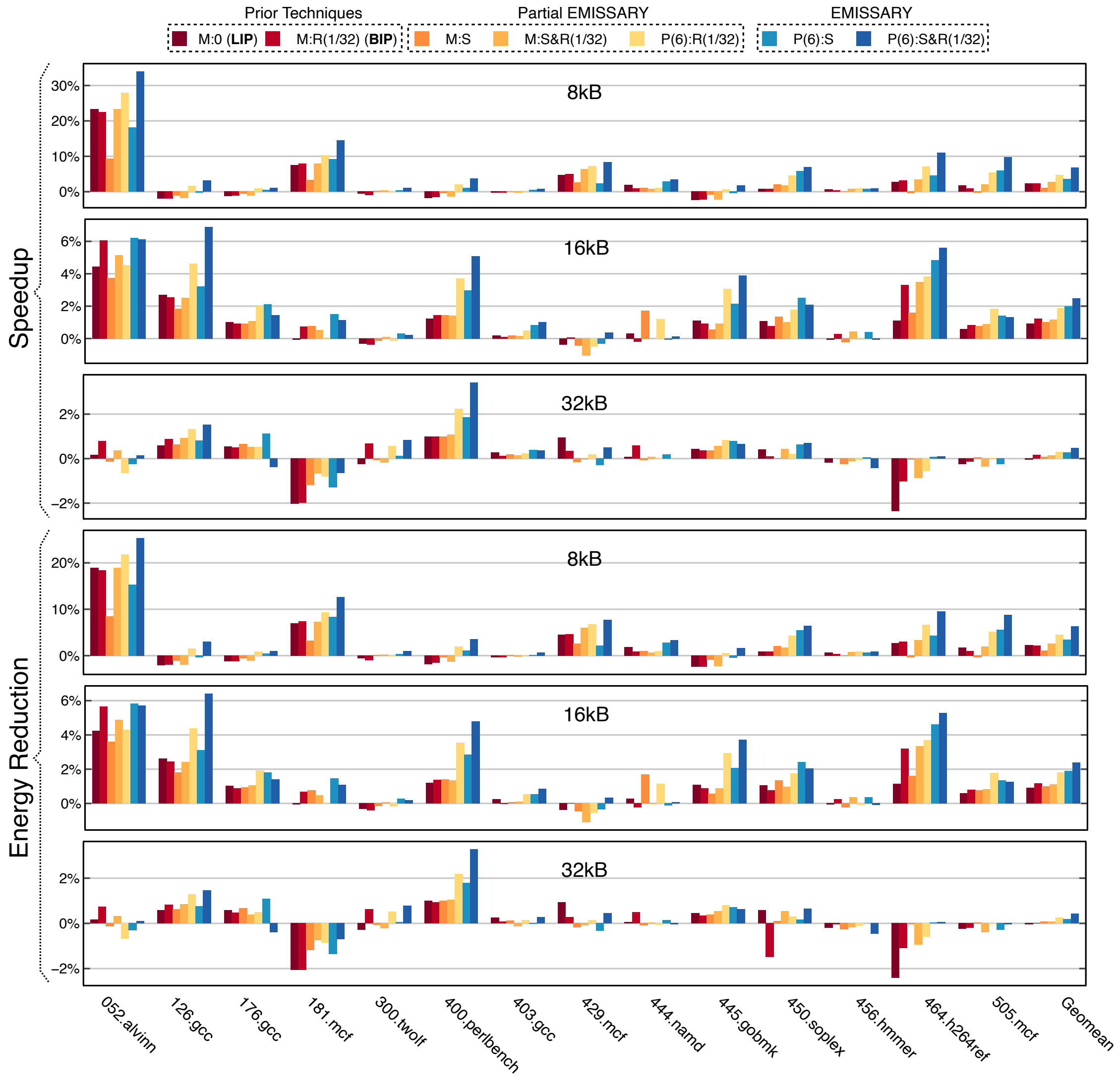


Figure 4.23: Speedup and Energy Reduction of a range of techniques relative to LRU, when simulated with the 'Default Gem5' Model for the first 100 million instructions of each program

Program	L1I size	Speedup over OPT
300.twolf	8kB	0.79%
444.namd	16kB	0.03%
253.perlbmk	32kB	0.49%
445.gobmk	32kB	0.77%

Table 4.8: Contexts in which P(6):S&R(1/32) beats OPT.

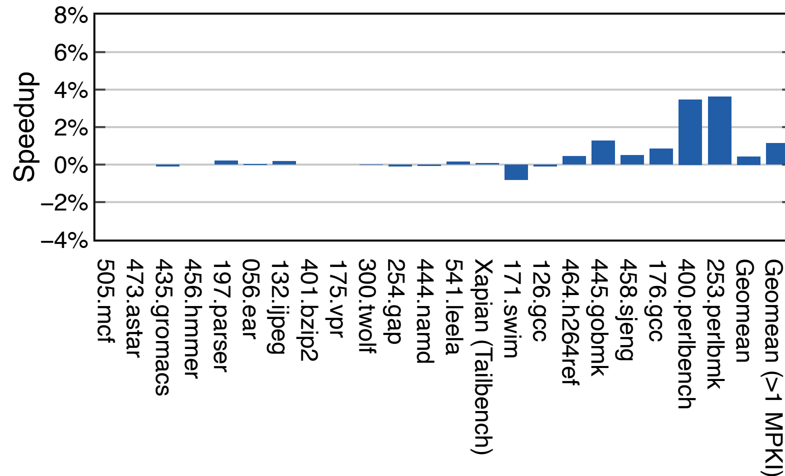


Figure 4.24: Speedup of EMISSARY (P(6):S&R(1/32)) relative to LRU depicted for all the programs when run on architecture with 32kB L1I cache, 8k BTB entries, and 20 FTQ entries.

when combining persistence, starvation, and the random filter.

Overall, P(6):S&R(1/32), the preferred EMISSARY configuration, yields geomean speedups of 2.1% for an 8K L1I, 2.6% for 16K, and 1.5% for 32K on the set of programs with an LRU MPKI (misses per kilo instructions) greater than 1.0. These numbers are significant given how aggressively these modeled architectures fetch instructions early guided by a sophisticated branch predictor and how often they tolerate L1I misses when they do occur. Additionally, this EMISSARY configuration also outperforms Bélády's OPT in a few cases (Table 4.8 lists the program, configuration, and the speedup achieved *over OPT* in each case).

Figure 4.24 presents the speedups achieved by EMISSARY (P(6):S&R(1/32)) when run on an architecture with larger front-end structures, in parallel with [25]. It includes

32kB L1I, 8k entries BTB, and 20 entries FTQ. Even with the state-of-the-art front-end, EMISSARY performs close to the results presented in the figure 4.22, with a minimal reduction in geomean to 1.2%. Such stability in EMISSARY's performance is unlike other proposals that have been shown to perform poorly with larger structures [25].

The effectiveness of EMISSARY is evident across program phases. Figure 4.25 presents the speedups achieved by the preferred EMISSARY configuration (P(6):S&R(1/32)) across the entire program duration for 458.sjeng and 541.leela, when run on a system with an 8kB L1I. We collected checkpoints using Lapidary [65] with a fixed interval, and at each checkpoint, we performed a detailed simulation for 100 million instructions. For most of the 541.leela execution, EMISSARY consistently achieves a speedup of more than 4%. With 458.sjeng, on the other hand, there are few regions where EMISSARY achieves speedup higher than 5%, up to nearly 10%.

4.3.5 Energy Savings

We used McPAT [42] to model the energy consumption of different cache replacement policies explored. Figures 4.22 and 4.23 show energy savings for each benchmark and configuration. The energy savings are strongly correlated with the speedups achieved because of the relatively small amount of hardware added. In EMISSARY, there are only two bits added per cache line, one to mark the priority set once on insert and an additional one for PLRU set on access. The EMISSARY P(6):S&R(1/32) configuration achieves a geomean reduction in overall energy of 2.0% at 8kB L1I, 2.5% at 16K, and 1.4% at 32K on the set of programs with an LRU MPKI greater than 1.0.

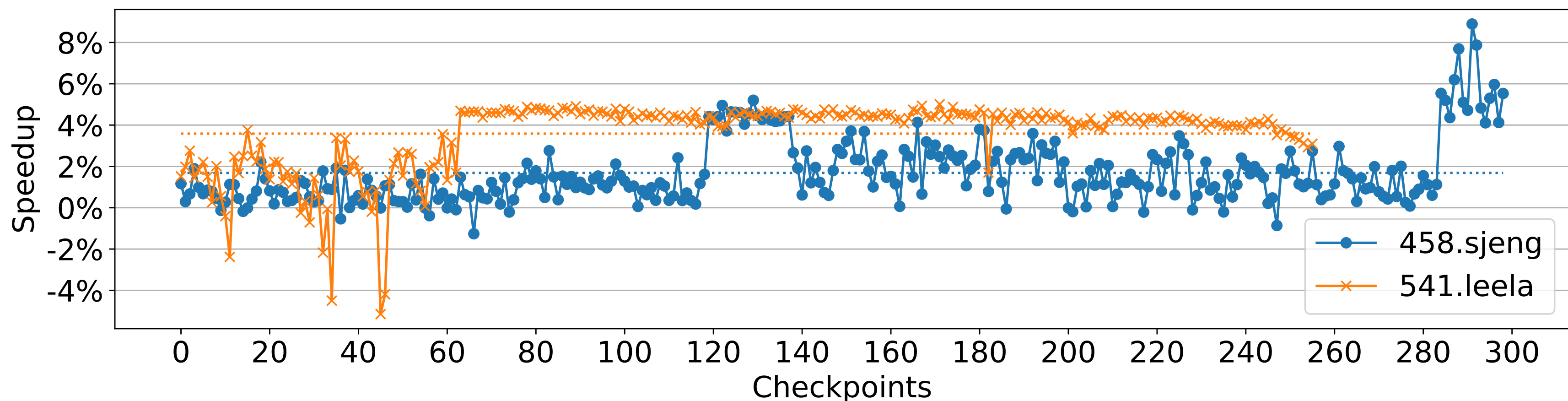


Figure 4.25: Speedup of EMISSARY (P(6):S&R(1/32)) relative to LRU depicted at each checkpoint, collected with a fixed interval. Both the programs were run to completion on a system with an 8kB L1I cache. The average IPC is 1.7 and 3.6 for 458.sjeng and 541.leela, respectively, depicted as a dotted line in the respective color.

4.4 Mechanisms to Improve Performance When Caches Are Big

In cases where the working set of a program is significantly small than the size of an L1I cache (i.e. when MPKI is very low), compulsory misses start to dominate. With fewer conflict misses in every single cache set, any cache replacement policy would not have much impact. Moreover, in such cases other effects may dominate. For a program dominated by compulsory misses, an LRU cache will quickly use all ways in the set. Since EMISSARY reserves N ways for high-priority lines, compulsory misses may compete for fewer ways in the set. As a result, few programs (056.ear, 132.ijpeg, 197.parser, and 456.hmmer) experience some losses with EMISSARY , which can be seen in the figures 4.22 and 4.23. Although the losses are minimal (less than 1% in all cases), we have suggested few mechanisms below to address it.

- **Turn off EMISSARY when the MPKI is less than 1.0:** A rather simple technique to minimize the losses while retaining the benefits of EMISSARY (when MPKI is larger), is to revert to LRU when the MPKI is low. This could be easily performed, by not considering the 'P' bit (Priority bit) upon eviction, by the EMISSARY replacement policy. The downside of such a policy is that it would take certain amount of accesses to identify the priority addresses again when the the MPKI increases and EMISSARY is turned on.

- **Reduce N:** This dissertation promotes N value of 6 for EMISSARY replacement policies, as described in §4.3.3. However, in cases when the MPKI is low, reducing N would allocate more ways in each set for the compulsory misses. As can be seen in figures 4.4 - 4.21, in the cases when EMISSARY loses (132.ijpeg, 197.parser, 456.hmmer at 8kB L1I), the loss is lesser for smaller values of N.

- **Set Dueling:** One of the popular mechanism to minimize the losses among cache

replacement policies has been Set Dueling [28, 52]. In Set Dueling, few sets in the cache are randomly selected to be 'Leader' sets, and the all the remaining sets are then chosen to be 'Follower' sets. Half of the leader sets rely upon the baseline replacement policy, which in our case is LRU, and the other half of the leader sets rely upon the proposed policy, in our case it is EMISSARY. Number of misses encountered by each leader set is noted. At any given point in time competing policy is selected among the two groups of leader sets that has minimal misses, and the chosen policy is used for all the follower sets.

Set Dueling has been proven by many others to perform well in larger, shared caches. However, in our experiments Set Dueling further hurt the performance for EMISSARY. This is because, for L1 caches the total number of sets available is far fewer, and hence Set Dueling would require larger fraction of the sets to be assigned as leader to best represent the population. Not just that, the correlation among sets is very poor in the case of instruction cache than the data cache. As a result, Set Dueling isn't a viable option for EMISSARY .

- **Set Dynamic:** An alternative to Set Dueling is the dynamic selection of competing policy individually for each set. In this policy, every set first executes a training phase, where in the baseline replacement policy (LRU in our case) is selected for say 'M' number of accesses, followed by another 'M' accesses with the EMISSARY policy (P(6):S&R(1/32)). Number of high-cost misses (misses that would be marked as high-priority in a P(N):S&R(1/32) EMISSARY policy) observed in each case is recorded. The policy that leads to the minimum number of high-cost misses is then chosen for the respective set.

  Figure 4.26 shows this Set Dynamic EMISSARY policy with 'M' being 50K. The first bar for each program shows the EMISSARY policy (P(6):S&R(1/32)), and the second bar shows the Set Dynamic EMISSARY policy. Surprisingly, Set Dynamic

improves the performance for 456.hmmer, which experienced losses in EMISSARY. Similarly, losses observed by 197.parser and 132.ijpeg are almost nullified with Set Dynamic. Interestingly, Set Dynamic improves the performance further for 444.namd and Xapian, which have a baseline MPKI of nearly 35 and 15 respectively (fig.4.15 and fig.4.21). However, Set Dynamic does impact the performance slightly in many other programs that experienced speedup with EMISSARY (for example 176.gcc, 253.perlbmk, 400.perlbench, 464.h264ref etc.)

As the program behavior changes over time, the sets chosen to rely on EMISSARY could soon become LRU friendly and vice versa. Thus, another variation of Set Dynamic policy that incorporates the program behavior is when the training phase repeats often and resets the chosen policy for each set. Figure 4.27 shows this when the training phase repeats after every 500K accesses. This variation of Set Dynamic policy retains the benefits of single training when MPKI is low, and also minimizes the impact when MPKI is large. Although there is a small reduction in the geometric mean of all the programs with the Set Dynamic policy, it incurs negligible amount of losses in all cases.

4.5 A Closer Look

In this section, we study EMISSARY at a deeper level to understand interesting patterns observed. We use the SPEC program 400.perlbench on a 32kB cache with the P(7):S&H(8) EMISSARY configuration because it prominently shows effects present to some degree in all programs.

4.5.1 Speedup Without Starvation Reduction

This work has shown that there is a better correlation between starvations on the committed path and processor performance than between performance and MPKI. However, this

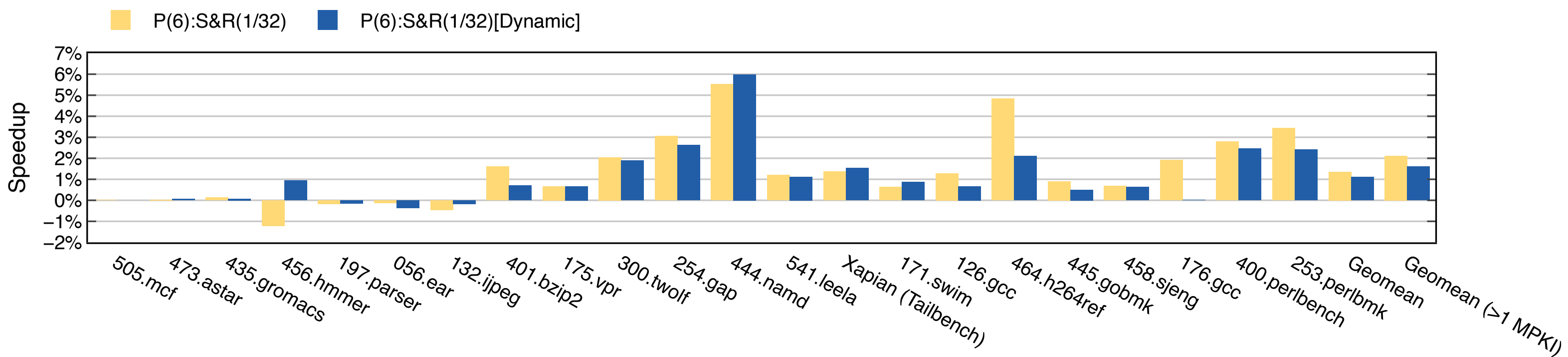


Figure 4.26: Set Dynamic Emissary with a single training phase of 50K accesses per set

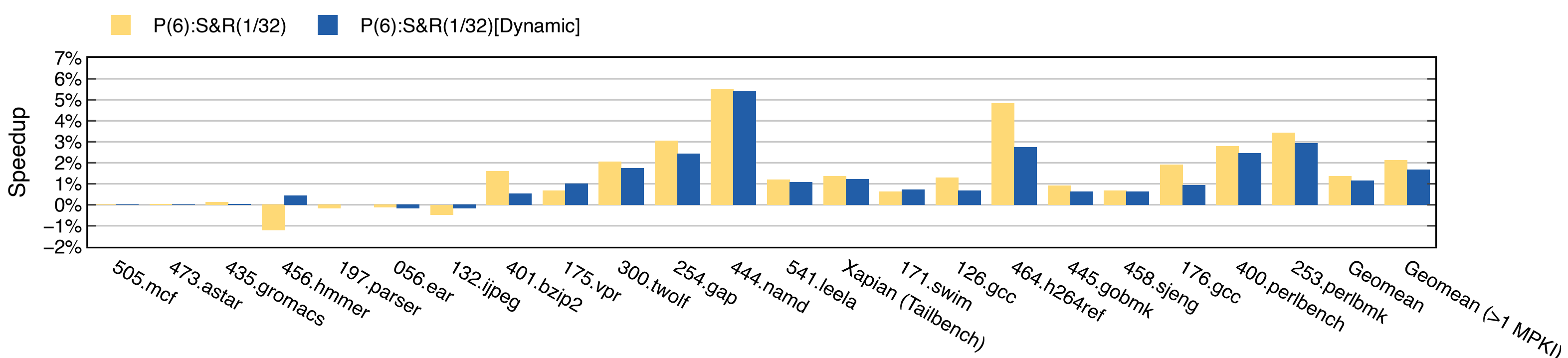


Figure 4.27: Set Dynamic Emissary with a training phase of 50K accesses per set repeating after 500K accesses

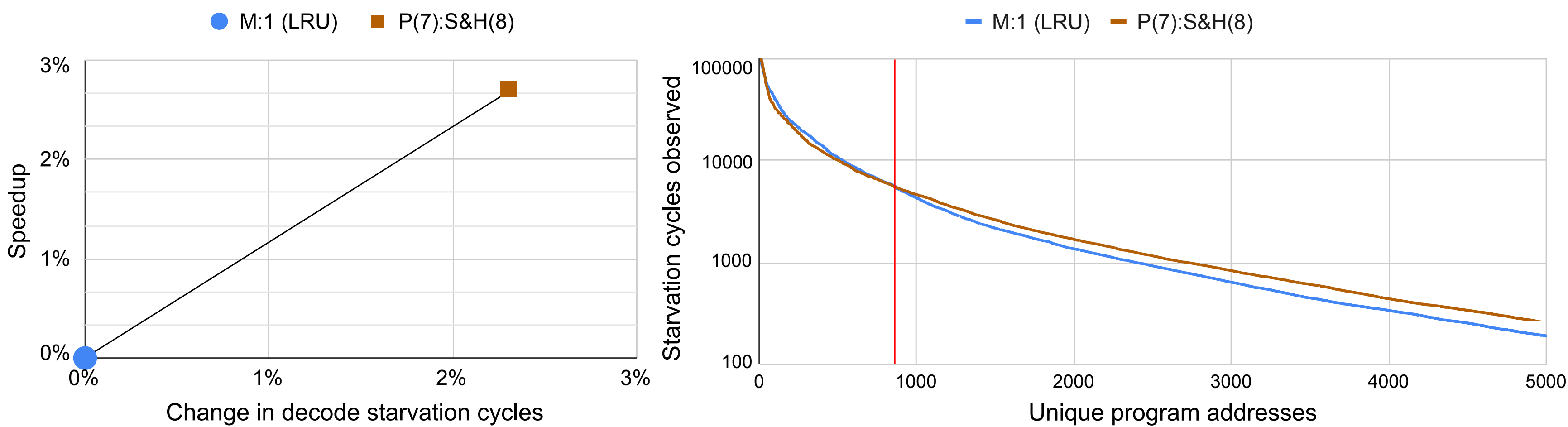


Figure 4.28: Performance, commit-path decode starvations, and the distribution of starvation cycles observed by each program address on the commit path of LRU and EMISSARY policy (P(7):S&H(8)) for 400.perlbench on a 32kB, 8-way L1I cache.

correlation does break down in some cases. Fig. 4.28 shows a negative correlation for the P(7):S&H(8) EMISSARY policy relative to LRU. (Note that this work does not promote the use of this EMISSARY policy because it requires relatively complex hardware to record starvation history and because it has an N of 7. However, this configuration prominently shows the effects highlighted in this section.) This configuration shows a performance gain of nearly 3%. At the same time, the number of decode starvations in the committed path increases by more than 2%. Although such a pattern does not occur as prominently in the common case, it is not unique to 400.perlbench. For example, as seen in Figures 4.4 - 4.21, 126.gcc, 176.gcc, 445.gobmk, and 464.h264ref at 32kB L1I have a slightly diverging trend line in the performance vs. starvation graphs.

All of the figures in this chapter report the starvation cycles encountered in only the path that eventually commits. This is because the number of starvations exposed in the mispeculated path does not ultimately affect the processor performance. It is surprising, then, to see higher performance even when the frontend pipeline stages experience a similar or increased number of starvations *in the committed path*.

As it turns out, just like misses, not all starvations have the same cost. Fig. 4.28 shows the distribution of the total number of starvation cycles observed by each program address when run on LRU and P(7):S&H(8). Note the logarithmic scale. The distribution shows:

- A smaller percentage of addresses accounts for a large fraction of the starvations.
- The EMISSARY policy more uniformly redistributes the starvations observed by addresses, shifting starvations by addresses that miss more frequently to the much larger set of addresses that starve less frequently.

The vertical red line at the x-axis value of 864 in Fig. 4.28 shows the transition in the starvation distribution trend from starvation reduction to starvation increase. Again, note the logarithmic scale. The total reduction in starvation cycles by the EMISSARY policy to the left of the split line is nearly 1.5 million, while the total addition in starvation cycles to the

right of the line is nearly 1.6 million.

The negative correlation observed in performance vs. starvation is more prevalent for larger values of N. The is because when there are fewer ways that can be used by low priority misses, the miss rate among the low-priority misses increases. Even lines with lower starvation rates starve, so with the increase in misses, starvation goes up too.

Thus, it is evident that reducing the starvations in decode is not the only way EMISSARY can improve performance. Starvation redistribution can improve performance by reducing stress on the backend. Uniformly distributed decode starvations are more likely to be tolerated than groups of starvations that deprive the backend of new instructions for longer periods of time. The EMISSARY techniques not only reduce starvations but also redistribute these starvations in a more backend-friendly manner.

4.5.2 Persistence, By Itself, Improves Hit Rate

Figures 4.4 - 4.21 shows that misses and processor performance are not directly correlated in some cases. It also shows that, in a majority of the programs and cache sizes, as N increases, L1I MPKI proportionately reduces. In other words, EMISSARY techniques do not just reduce starvation, but MPKI as well. Even as the number of ways available to a significant fraction of low-priority addresses is reduced, misses reduce as well. The reason is simple and observed previously with the BIP technique [52]. With the prevalence of single reference (or extremely long time between reference) addresses, dedicating fewer cache resources to such lines makes way for lines that would otherwise miss. In this aspect, starvation acts as a filter, increasing the probability of giving such lines low-probability and relegating them to a subset of available lines. Put another way, it helps reduce the extent to which these types of lines can pollute the cache.

4.6 Related Work

Owing to the increasing gap between the processor and the memory performance, improving the effectiveness of the caches, specifically cache replacement algorithms have been of interest to academia and industry for decades. The book on cache replacement policies [27] has extensively explored the space. This section describes several cache replacement strategies having similarities to EMISSARY.

4.6.1 Cost-Aware Cache Replacement Policies

Architects have long recognized that not all cache misses have the same performance cost. In light of this observation, several prior works have proposed cost-aware cache replacement policies that give deference to lines with higher miss costs while selecting a line to evict [29, 30, 35, 47, 53, 63]. There exists a huge spectrum of definitions for the miss cost, ranging from latency to number of instructions issued while a miss is served. Interestingly, all of these techniques target either data or shared caches. We are not aware of any proposed cost-aware cache replacement algorithm designed solely for instruction caches.

CSOPT [30] is the ideal cost-aware cache replacement algorithm. It is essentially Bélády's OPT augmented with cost awareness. Like OPT, CSOPT is unrealizable as it also requires knowledge of the future.

Among the realizable cost-aware policies, MLP-aware replacement policies [4, 53] identify costly misses by observing memory-level parallelism (MLP). These techniques reduce the overall miss cost by attempting to reduce the number of isolated misses (i.e., misses that do not have MLP). The MLP LIN policy utilizes the miss status handling register (MSHR) as input to cost calculation hardware that uses fixed point calculations. Additionally, quantized three cost bits are stored along with the tags for each cache entry. In contrast, there is no cost calculation in EMISSARY other than obtaining the starvation signal, a signal already present in the processor. And, unlike MLP LIN, EMISSARY stores just a single

cost bit per entry. Moreover, MLP LIN does not perform as well as LRU, their baseline, in many cases, and the suggested SBAR dynamic mechanism demands additional storage overhead. Although the idea of MLP LIN could be extended to instruction cache, it would not have the capability to distinguish the starvation-prone addresses from the ones that are due to the side effects of mispredicted branches, for example. EMISSARY includes the most simplistic, direct, and accurate form of cost identification, specifically for instruction caches, which cannot be approximated by any other means.

LACS [35] is a cost-aware cache replacement policy for last-level caches. It calculates the cost by counting the number of instructions that the processor can issue and execute while the miss is being serviced. After insertion, LACS uses reference information to adjust the priority. In contrast, EMISSARY doesn't modify the priority information after insertion. This is based on the observation that most of the instructions that cause decode starvation, do so every time they miss, and hence they benefit by being persistently categorized as high-priority. Likewise, the instructions that don't cause decode starvation, would consistently not cause starvation on every miss, and hence they wouldn't need to be categorized as high priority even if they are frequently accessed. Like EMISSARY, LACS requires two bits per line. Unlike EMISSARY, LACS also requires a history table, counters, and enhancements to the MSHRs.

Critical Cache [63] proposes the use of a victim cache to give performance critical loads a second chance. Critical Cache identifies critical loads using heuristics related to the types of instructions executing near the load. Despite the extra hardware, the Critical Cache paper does not report performance gains over LRU.

4.6.2 Policies Challenging LRU

Even putting cost-aware considerations aside, one considerable contribution to the level of performance that is achieved by EMISSARY is the fact that cache lines do not have their priorities changed by subsequent references (subsequent references always play a secondary

role in replacement). This distinguishes it from LRU [44], which assigns the most recently used line the highest priority in the cache. This is important to note because, as was seen as a result of adding randomization, simply adding any priority filter [52], a bar which addresses must meet before considering them important, is often enough to improve performance over LRU. This idea of trying to keep important cache lines in the cache while excluding others has been seen in a number of forms, such as the commonly known Least Frequently Used (LFU) replacement policy [13], various cache bypassing techniques [36, 45], and others [33, 40, 60].

In prior work, lines not meeting a bar may still be inserted but in a way that has them closer to eviction [52]. Such techniques proposed always inserting the lines into the LRU position to improve performance in the case of highly thrashing working sets (LRU Insertion Policy, LIP). The policy was further improved upon with a schema that only inserted newly incoming lines into the LRU slot at a low, random probability (Bimodal Insertion Policy, BIP). The final modification was a dynamic method, which chose between traditional LRU and the aforementioned random insertion policy, depending on which resulted in fewer misses for a given workload (Dynamic Insertion Policy, DIP). The parallel between this cache replacement policy and EMISSARY is that both decide to give preference to some lines over others to keep in the cache based on a factor other than recency. The key difference, though, is EMISSARY sets the bar with a relevant cost signal in addition to randomness. Importantly, EMISSARY persists the priority information for the line's lifetime. It is in this way that EMISSARY not only benefits from requiring lines to prove themselves as being commonly used before being deemed important to keep in the cache, but also from making this determination based on a specific cost factor that proves to be effective consistently for given workloads.

GHRP [1], an instruction cache replacement policy focused on minimizing the number of misses by identifying dead-blocks, is orthogonal to ours. They use the access history to identify if a block needs to be bypassed upon insertion. Else, the dead-block predictor is

used to select the candidate for eviction. GHRP's dead-block prediction mechanism could be combined with EMISSARY to identify the low-priority dead blocks for eviction. Doing so might further improve the performance of EMISSARY .

4.6.3 OPT Inspired Strategies

Another theme that is practiced in a number of other proposed cache replacement policies is having cache lines' past history help to influence future cache insertion and/or eviction decisions [3, 18, 26, 48, 52, 67]. Specifically, in the case of [52], the past access patterns are used to determine a mode of a given dynamic replacement policy. It is also seen in [26, 28, 43, 52], where access history is recorded and used to best emulate a particular cache replacement policy. The most common policy to emulate is Bélády's Algorithm, and it is proven to be optimal in terms of minimizing cache misses. This cache replacement policy, while optimal, is impossible to implement in practice because it requires knowledge of the future to inform eviction decisions. Nonetheless, given that this algorithm is in some respects the asymptotic cache replacement limit, many cache replacement algorithms have sought to emulate it as closely as possible using its own past access history and learning from it. One such replacement policy that aims to do this is Hawkeye [26]. Hawkeye implements Bélády's Algorithm on past cache access patterns that are encountered for a given program, and as the algorithm progresses, memory from the decisions that are made in terms of evictions with respect to the lines' surrounding are recorded and learned. This recording and learning is translated into a PC-based binary classifier. The important distinction that EMISSARY makes from a policy like this is that our replacement policy does not seek primarily to reduce cache misses, but rather its focus is on performance. This breaks the asymptotic restriction of Bélády's Algorithm for cache replacement policies as there is no implied inextricable link between cache hit rate and overall processor performance.

Another cache replacement policy that takes its inspiration from the optimal cache replacement algorithm is Re-Reference Interval Prediction [28]. This replacement algorithm

focusses on re-reference patterns of cache lines and improves upon LRU by taking advantage of the fact that LRU performs poorly when the re-reference patterns of lines are not near immediate, as in the case of workloads that cause high rates of thrashing or scanning (single-use accesses). The key insight this algorithm provides is that it allows for an effective strategy of dealing with these varying workloads. EMISSARY similarly outperforms LRU in terms of cache misses owing to the fact that it does not unconditionally give cache preference to the most recently used lines. EMISSARY treats lines differently based on the usage history, and in this way, it is similar to this re-reference based policy. The difference, however, is the key insight EMISSARY makes that influences its decision-making on starvations, rather than purely re-reference times. Our design aims at solving a more general use case problem than one that is focused on specific workloads that contain large working sets or a high number of single-use cache lines that lead to thrashing or cache pollution under LRU.

Chapter 5

Concluding Remarks and Future Directions

Owing to the increasing reliance on cloud computing, it is more critical now than earlier to efficiently manage the data center CPU architecture. Even the slightest improvement in the performance of the data center processors leads to significant cost margins and energy savings.

Despite enormous software optimizations and increasing hardware complexity, our work with Google shows tremendous CPU usage inefficiencies in WSCs. Specifically, nearly 20% of the wastages are due to the inefficiencies in the front-end of the processor. Since the processor front-end is responsible for keeping the pipeline fed, it dictates the maximum achievable performance limit. Thus, it is crucial to increase the efficiency of the front-end to improve the overall CPU efficiency.

As most of the front-end bottlenecks are due to instruction cache inefficiencies, in this Dissertation, we have studied in-depth the root causes for instruction cache misses in a WSC, described in Chapter 2. Our longitudinal study spanning the breadth of Google's fleet shows significant code fragmentation at both intra- and inter-cache-line granularity, leading to inefficient instruction cache usage. We have also shown that most of the misses are due to

distant branch and call instructions.

Based on our observation, in Chapter 3, we have demonstrated that not all instruction cache misses have the exact performance cost. Additionally, we have shown that decode starvation is the most direct form of identifying the costly misses.

In Chapter 4, we have presented EMISSARY , the first-of-its-kind cost-aware instruction cache replacement policy. Basing upon the decode starvation signal, EMISSARY amazingly learns the most expensive instruction cache addresses. Furthermore, we show that the replacement policy should also persist the priority information. Despite the fact that modern processors completely tolerate first-level instruction cache misses, EMISSARY shows up to 6% performance improvement on few large workloads. Its simplicity, combined with its stability and energy efficiency benefits, makes it a profitable choice for future processor designs.

5.1 Future work

This Dissertation has presented and evaluated EMISSARY in the context of L1 instruction caches. EMISSARY can, however, might be valuable in other caches found in modern processors – data caches, second-, and higher-level caches, Branch Target Buffer (BTB), instruction Translation Lookaside Buffer (TLB), and data TLB. Testing the effectiveness of EMISSARY in all these other caches or cache-like structures is left to future work. We will briefly discuss some of the design considerations for different cache categories below.

Since BTB and instruction-TLB are part of the processor front-end, the decode starvation signal could be used as a viable cost-metric input. However, the number of ways needed for high-priority entries might vary because of the different associativity found in these structures. Additionally, all these front-end structures – instruction caches, BTB, instruction TLB, and even Branch Predictors will have to be considered holistically while designing a well-performant front-end.

In contrast, a different cost-metric will have to be used to identify costly misses on the data cache side. There is a vast spectrum in this space, and the EMISSARY technique would be susceptible to the cost-identification method chosen. We tested one cost-calculation method as an initial choice based on the number of issue idle cycles. Our cost-calculation logic marked an L1D miss as a high-priority one if the respective miss led to at least three clock cycles of no issue period. Although our initial results with such a cost-calculation metric did not show clear patterns as we observed with L1I cache, this EMISSARY policy and many other variations with different cost-metrics are worth to be explored further.

When it comes to L2 and other higher-level shared caches, they need a different cost-identification metric, as these caches are shared between instructions and data and even between different cores. EMISSARY policy, in that case, will also have to take into account the behavior of lower-level caches. One straightforward possibility is to track the costly misses in lower-level caches (L1I and L1D), which are marked similarly in higher-level caches. Another variation would be to have a different threshold for costly misses in different cache levels. For example, longer duration of decode starvation cycles if on instruction-path, or longer duration of issue idle cycles if on data-path can be set as a threshold to identify the costly misses marked in the L2 cache. As it can be seen, the space of exploration quickly expands when it comes to higher-level caches, and it is left for future work.

Bibliography

[1] S. M. Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00050

[2] Akamai, "Akamai online retail performance report: Milliseconds are critical," https://www.ir.akamai.com/news-releases/news-release-details/akamai-online-retail-performance-report-milliseconds-are.

[3] E. R. Altman, V. K. Agarwal, and G. R. Gao, "A novel methodology using genetic algorithms for the design of caches and cache replacement policy." in *ICGA*, 1993, pp. 392–399. [Online]. Available: https://doi.org/10.1109/WorldS450073.2020.9210347

[4] A. Arunkumar and C.-J. Wu, "Remap: Reuse and memory access cost aware eviction policy for last level cache management," in *2014 IEEE 32nd Annual Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 110–117. [Online]. Available: https://doi.org/10.1109/iccd.2014.6974670

[5] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *High Performance Computer Architecture (HPCA)*, 2018. [Online]. Available: https://doi.org/10.1109/HPCA.2018.00061

[6] G. Ayers, N. P. Nagendra, 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," in *Computer Architecture (ISCA)*, 2019. [Online]. Available: https://doi.org/10.1145/3307650.3322234

[7] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The datacenter as a computer: Designing warehouse-scale machines*. Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2018. [Online]. Available: https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[8] L. Belady, "A study of replacement algorithms for a virtual-storage computer," in *IBM Systems journal, pages 78–101*, 1966. [Online]. Available: https://doi.org/10.1147/sj.52.0078

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[10] "Internet statistics," https://www.broadbandsearch.net/blog/internet-statistics.

[11] G. Chatelet, C. Kennelly, S. Xi, O. Sykora, C. Courbet, D. Li, and B. D. Backer, "automemcpy: A framework for automatic generation of fundamental memory operations," in *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, 2021. [Online]. Available: https://doi.org/10.1145/3459898.3463904

[12] D. Chen, D. X. Li, and T. Moseley, "Autofdo: Automatic feedback-directed optimization for warehouse-scale applications," in *Code Generation and Optimization (CGO)*, 2016. [Online]. Available: https://doi.org/10.1145/2854038.2854044

[13] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Prentice-Hall, 1973.

[14] R. Cohn and P. G. Lowney, "Hot cold optimization of large windows/nt applications," in *Microarchitecture (MICRO)*, 1996. [Online]. Available: https://doi.org/10.1109/MICRO.1996.566452

[15] "Global digital overview," https://datareportal.com/global-digital-overview.

[16] O. Djuraskovic, "Google search statistics," https://firstsiteguide.com/google-search-stats/.

[17] Y. Environment, "Yale environment," https://e360.yale.edu/features/energy-hogs-can-huge-data-centers-be-made-more-efficient.

[18] P. Faldu and B. Grot, "Leeway: Addressing variability in dead-block prediction for last-level caches," in *In Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*, 2017, pp. 180–193. [Online]. Available: https://doi.org/10.1109/pact.2017.32

[19] M. Ferdman, B. Falsafi, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki, "Clearing the clouds," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012. [Online]. Available: https://doi.org/10.1145/2150976.2150982

[20] Forbes, "Forbes report on expected iot trends in 2021," https://www.forbes.com/sites/danielnewman/2020/11/25/5-iot-trends-to-watch-in-2021/?sh=72d1c31d201b.

[21] U. Gupta, Y. G. Kim, S. Lee, J. Tse, H.-H. S. Lee, G.-Y. Wei, D. Brooks, and C.-J. Wu, "Chasing carbon: The elusive environmental footprint of computing," in *arXiv:2011.02839*, 2020. [Online]. Available: https://arxiv.org/abs/2011.02839

[22] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., 2012.

[23] ——, “A new golden age for computer architecture,” *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019. [Online]. Available: https://doi.org/10.1145/3282307

[24] G. J. Hinton and R. M. Riches Jr, “Instruction fetch unit with early instruction fetch mechanism,” Jun. 6 1995, uS Patent 5,423,014. [Online]. Available: https://patents.google.com/patent/US5423014A/en

[25] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, “Rebasing instruction prefetching: An industry perspective,” *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 147–150, 2020. [Online]. Available: https://doi.org/10.1109/LCA.2020.3035068

[26] A. Jain and C. Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” in *43rd International Symposium on Computer Architecture (ISCA)*, 2016. [Online]. Available: https://doi.org/10.1109/ISCA.2016.17

[27] ——, *Cache Replacement Policies*. Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2019. [Online]. Available: https://doi.org/10.2200/S00922ED1V01Y201905CAC047

[28] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. Emer, “High performance cache replacement using re-reference interval prediction (rrip),” in *37th International Symposium on Computer Architecture (ISCA)*, 2010. [Online]. Available: https://doi.org/10.1145/1815961.1815971

[29] J. Jeong and M. Dubois, “Cost-sensitive cache replacement algorithms,” in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* IEEE, 2003, pp. 327–337. [Online]. Available: https://doi.org/10.1109/HPCA.2003.1183550

[30] ——, “Cache replacement algorithms with nonuniform miss costs,” *IEEE Transactions on Computers*, vol. 55, no. 4, pp. 353–365, 2006. [Online]. Available: https://doi.org/10.1109/TC.2006.50

[31] Juniper, "Juniper research on iot growth," https://www.juniperresearch.com/press/iot-connections-to-reach-83-bn-by-2024.

[32] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Computer Architecture (ISCA)*, 2015. [Online]. Available: https://doi.org/10.1145/2749469.2750392

[33] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," in *Computer*, 1994, pp. 38–46. [Online]. Available: https://doi.org/10.1109/2.268884

[34] H. Kasture and D. Sanchez, "TailBench: A benchmark suite and evaluation methodology for latency-critical applications," in *Workload Characterization (IISWC)*, 2016. [Online]. Available: https://doi.org/10.1109/IISWC.2016.7581261

[35] M. Kharbutli and R. Sheikh, "Lacs: A locality-aware cost-sensitive cache replacement algorithm," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 1975–1987, 2014. [Online]. Available: https://doi.org/10.1109/TC.2013.61

[36] M. Kharbutli and Y. Solihin, "Counter-based cache replacement and bypassing algorithms," *IEEE Transactions on Computers*, vol. 57, no. 4, pp. 433–447, 2008. [Online]. Available: https://doi.org/10.1109/TC.2007.70816

[37] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018. [Online]. Available: https://doi.org/10.1145/3173162.3173178

[38] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *High Performance Computer Architecture (HPCA)*, 2017. [Online]. Available: https://doi.org/10.1109/HPCA.2017.53

[39] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 2, pp. 59–70, 2002. [Online]. Available: https://doi.org/10.1145/545214.545223

[40] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," in *IEEE Transactions on Computers*, 2001, pp. 1352–1361. [Online]. Available: https://doi.org/10.1109/tc.2001.970573

[41] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, and T. B. Schardl, "There's plenty of room at the top: What will drive computer performance after moore's law?" *Science*, vol. 368, 2020. [Online]. Available: https://doi.org/10.1126/science.aam9744

[42] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480. [Online]. Available: https://doi.org/10.1145/1669112.1669172

[43] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *arXiv:2006.16239*, 2020. [Online]. Available: https://arxiv.org/abs/2006.16239

[44] R. L. Mattson, J. Gegsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," in *IBM Systems Journal*, 1970, pp. 78–117. [Online]. Available: https://doi.org/10.1147/sj.92.0078

[45] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016. [Online]. Available: https://doi.org/10.1145/2893356

[46] 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*, vol. 40, no. 3, pp. 56–63, 05 2020. [Online]. Available: https://doi.org/10.1109/MM.2020.2986212

[47] A. V. Nori, J. Gaur, S. Rai, S. Subramoney, and H. Wang, "Criticality aware tiered cache hierarchy: a fundamental relook at multi-level cache hierarchies," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 96–109. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00019

[48] M. Palmer and S. B. Zdonik, *Fido: A cache that learns to fetch*. Brown University, Department of Computer Science, 1991. [Online]. Available: http://vldb.org/conf/1991/P255.PDF

[49] X. Pan and B. Jonsson, "A modeling framework for reuse distance-based estimation of cache performance," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 62–71. [Online]. Available: https://doi.org/10.1109/ISPASS.2015.7095785

[50] M. Panchenko, R. Auler, B. Nell, and G. Ottoni, "BOLT: A Practical Binary Optimizer for Data Centers and Beyond," in *Code Generation and Optimization (CGO)*, 2019. [Online]. Available: https://doi.org/10.1109/CGO.2019.8661201

[51] T. R. Puzak, "Analysis of cache replacement-algorithms," PhD dissertation, University of Massachusetts Amherst, 1986. [Online]. Available: https://scholarworks.umass.edu/dissertations/AAI8509594/

[52] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. Emer, “Adaptive insertion policies for high performance caching,” in *34th International Symposium on Computer Architecture (ISCA)*, 2007. [Online]. Available: https://doi.org/10.1145/1273440.1250709

[53] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, “A case for mlp-aware cache replacement,” in *33rd International Symposium on Computer Architecture (ISCA’06)*. IEEE, 2006, pp. 167–178. [Online]. Available: https://doi.org/10.1109/ISCA.2006.5

[54] G. Reinman, T. Austin, and B. Calder, “A scalable front-end architecture for fast instruction delivery,” *SIGARCH Comput. Archit. News*, vol. 27, no. 2, p. 234–245, May 1999. [Online]. Available: https://doi.org/10.1145/307338.300999

[55] G. Reinman, B. Calder, and T. Austin, “Fetch directed instruction prefetching,” in *Microarchitecture (MICRO)*, 1999. [Online]. Available: https://doi.org/10.1109/MICRO.1999.809439

[56] ——, “Optimizations enabled by a decoupled front-end architecture,” *Computers, IEEE Transactions on*, vol. 50, pp. 338 – 355, 05 2001. [Online]. Available: https://doi.org/10.1109/12.919279

[57] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, “Google-wide profiling: A continuous profiling infrastructure for data centers,” *IEEE Micro*, 2010. [Online]. Available: https://doi.org/10.1109/MM.2010.68

[58] D. Sanchez and C. Kozyrakis, “Zsim: fast and accurate microarchitectural simulation of thousand-core systems,” in *Computer Architecture (ISCA)*, 2013. [Online]. Available: https://doi.org/10.1145/2485922.2485963

[59] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Higher Education, 2004.

[60] Y. Smaragdakis, S. Kaplan, and P. Wilson, “Eelru: Simple and effective adaptive page replacement,” in *ACM SIGMETRICS Performance Evaluation Review*. ACM, 1999, pp. 122–133. [Online]. Available: https://doi.org/10.1145/301464.301486

[61] SPEC, “Spec newsletter,” ”https://www.spec.org”. [Online]. Available: https://www.spec.org

[62] ——, “Spec 2006 benchmark descriptions,” 2006. [Online]. Available: https://www.spec.org/cpu2006/CFP2006/

[63] S. T. Srinivasan, R. D.-C. Ju, A. R. Lebeck, and C. Wilkerson, “Locality vs. criticality,” in *Proceedings 28th Annual International Symposium on Computer Architecture*. IEEE, 2001, pp. 132–143. [Online]. Available: https://doi.org/10.1145/379240.379258

[64] T. Way and L. Pollock, “Evaluation of a region-based partial inlining algorithm for an ILP optimizing compiler,” in *Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2002.

[65] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, “Nda: Preventing speculative execution attacks at their source,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 572–586. [Online]. Available: https://doi.org/10.1145/3352460.3358306

[66] W. S. Wong and R. J. T. Morris, “Benchmark synthesis using the lru cache hit function,” *IEEE Transactions on Computers*, vol. 37, no. 6, pp. 637–645, 1988. [Online]. Available: https://doi.org/10.1109/12.2202

[67] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, J. Simon C. Steely, and J. Emer, “Ship: Signature-based hit predictor for high performance caching,” in *In 44th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 430–441. [Online]. Available: https://doi.org/10.1145/2155620.2155671

[68] A. Yasin, “A Top-Down method for performance analysis and counters architecture,” in *Performance Analysis of Systems and Software (ISPASS)*, 2014. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844459