Instruction Presending

by

Shyam Murthy

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2024

Date of final oral examination: 07/26/2024

Dedicated to my parents and my brother, for their unconditional love and support.

Acknowledgements

First and foremost, I would like to thank my advisor, Guri Sohi for taking me on as his student and for the financial support throughout my graduate studies. He has significantly influenced my way of thinking about technical problems and his training is now an inseparable part of me as a researcher. Among the multiple attributes I have imbibed from him, I distinctly recall his attention to detail and ever growing definition of what is experimentally possible. Most importantly, he forced me to develop my own ideas. It has indeed been a privilege working for him.

Next, I would like to express my gratitude to Mikko Lipasti, who has been incredibly generous with his time. I have gained extensive knowledge in computer architecture, microarchitecture, and more through our discussions. His feedback at various stages of my research has been invaluable. I would also like to thank the other members of my thesis committee, Karu Sankaralingam, Jignesh Patel, and Joshua San Miguel, for their critical questions during my preliminary exam and for helping to elevate the quality of my work.

I extend my thanks to former members of this group, Hongil Yoon and Gagan Gupta, for their invaluable advice during the early stages of my graduate program. I am also grateful to Andreas Moshovos for reviewing our work, providing feedback, and assisting with my job search. Special thanks go to my office mate Swapnil Haria for helping me get up to speed as a new graduate student.

I would also like to thank the regulars of the Computer Architecture Reading group—Swapnil Haria, Gokul Subramanian, Kyle Daruwalla, Ravi Raju, Chien-Fu Chen, Heng Zhuo, Bujji Selagamsetty, Rutwik Jain, and Vishnu Ramadas—for many engaging paper discussions. Additionally, I appreciate the insightful conversations with students from the Computer Architecture Research Group and Systems Research Group over the past five years.

I would also like to thank Amanda Xu, Abtin Molavi, Eric Pauley, Ryan Sheatsley, Kunyang Li, Yohan Beugin, and Satwik Maurya for the wide-ranging discussions, from research to philosophy, that we have shared over the past two years.

My stay in Madison was also made enjoyable with new friends: Arjun Balasubramanian, Rohit Sharma, Giri Prasanna, Mohammad Danish, Aditya Rungta, Zhiwei Fan, Zifan Liu, Xinyu Guan and many more.

I would also like to extend my heartfelt thanks to my family friends around the Chicago area for their warm hospitality and for generously hosting me on multiple occasions. Their home-cooked meals were the best I could ask for, and their kindness made me feel truly at home.

Last but not least, I would like to express my heartfelt gratitude to my parents and my brother for their unwavering love and support. They made countless sacrifices to ensure I received the best education, even when it meant we had to be apart. I am especially thankful to my mother for the many conversations we've shared over the years. I also wish to acknowledge my late grandparents, whose influence has greatly shaped the person I am today.

I tender my apologies in advance to all those whose mention has been inadvertently forgotten.

Contents

List of Tables

List of Figures

Abstract

Modern software development practices leads to the production of applications with increasingly large code footprints. This growth is driven by the emphasis on maintaining functionality, portability, and maintainability of software. The large code footprint begins to overwhelm primary structures such as the L1 instruction cache (L1i), Instruction TLB (iTLB), and Branch Target Buffer (BTB) in a processor that are crucial for ensuring an effective supply of instructions for execution. This results in frequent movement of code cache blocks and other closely related information, such as iTLB and BTB entries, between secondary and primary structures. To address this challenge, this dissertation proposes a different way to proactively move instructions and closely related information from secondary structures to the primary structures before the processor references them. The key technique introduced is:

Instruction Presending: This technique leverages stability in high-level control flow and the capability to capture this information in practical hardware tables to resolve upcoming high-level control flow. It runs ahead of the processor, resolving high-level control flow and determining cache blocks of instructions likely to be referenced by the processor, and moving them to the L1i before the processor references them. Furthermore, this technique is enhanced to support **Presending for an iTLB/BTB**, utilizing the high-level sequencing mechanism to identify the iTLB/BTB entries likely to be referenced by the processor and proactively move them to the iTLB/BTB before the processor requires them.

Instruction Presending shows significant promise in accurately resolving upcoming high-level control flow for many applications, particularly those with fre-

quent code movement. This results in a near-perfect determination of cache blocks of instructions and iTLB/BTB entries for these applications, leading to performance that approaches that of a perfect L1i/iTLB/BTB in these scenarios.

Chapter 1

Introduction

Executing a program may initially appear to be a highly sequential process: (1) Fetch an instruction, (2) Read input data values for the instruction, (3) Store the output of the instruction if necessary, and (4) Decide which instruction to fetch next. However, researchers have observed that substantial amounts of *Instruction-Level Parallelism* (*ILP*) exist within instruction streams [12, 19]. This means that instructions later in the stream can execute in parallel with earlier instructions, as they are part of independent computations. Modern superscalar processors [60, 65] have successfully transitioned from a sequential model of processing instructions to a highly parallel model, extracting high amounts of ILP.

Extracting these higher levels of instruction-level parallelism necessitates establishing true dependencies between instructions using techniques such as register renaming [72] and memory dependence prediction [26, 53, 54]. The last few decades have seen several innovations aimed at facilitating increasing levels of performance for these ILP machines. All these techniques rely on the creation of large windows of instructions to enable this parallel execution. The creation of large instruction windows is necessary to keep the backend of the machine— which extracts ILP—well-supplied with instructions. Inefficiencies in the frontend, which is responsible for supplying these instructions, result in reduced ILP.

This process of supplying instructions to the backend involves going to memory to fetch instructions every cycle. There is a large discrepancy between processor speeds and latency of accessing instructions from memory off-chip, referred to as the memory wall [81, 82]. This pushed processor designs to build memory hierarchies with multiple levels of caches on-chip to satisfy the bandwidth needs of the processor and to enable faster access times, exploiting the spatial and temporal locality present in the reference streams. This provides an illusion of a faster and larger memory at the cost of a slower and cheaper memory. The goal is to have all active instructions be in the L1 instruction cache (L1i) when needed, operating at close to processor speeds and satisfying the bandwidth needs of the processor.

Virtual memory provides each process with the illusion of a large private address space and it is a ubiquitous abstraction provided to programmers today. Virtual memory using paging is common in most computer systems of today. Implementing an efficient virtual memory system involves software and hardware involvement. Virtual address space is divided into coarse-grained, fixed-size chunks called pages which are mapped to physical frames. This mapping typically happens via a page table or a hierarchy of page tables, which is set up by software (or the operating system (OS)).

Every instruction memory reference generates a virtual address and has to be translated from a virtual to a physical address. Memory accesses to the page table (or hierarchy) made for every reference will slow the system down due to the memory wall. To mitigate this issue, nearly all modern computers utilize a hardware cache known as the Translation Lookaside Buffer (TLB). This cache stores recently accessed address translations, accelerating the translation process to meet the processor's bandwidth requirements. The TLB exploits both temporal and spatial locality within the instruction memory reference stream. The goal is to have all active translations for instructions be present in the Instruction Translation Lookaside Buffer (iTLB) when needed. This setup enables operations at close to processor speeds, effectively meeting the processor's bandwidth demands.

Instruction streams often experience discontinuities in the sequential flow of instructions due to programming constructs such as if-then-else statements, loops, and function calls. The address to fetch instructions from, following a discontinuity, is not known during instruction fetch but is determined later in the instruction pipeline. Processors typically have tables to record these discontinuities and logic to appropriately redirect the flow of instructions past future instances of these discontinuities, ensuring a sustained flow of instructions to the backend. A common structure within the processor employed for this purpose is the *Branch Target Buffer* (*BTB*). The goal is to have the BTB accommodate all active discontinuities, effectively meeting the demands of the processor and ensuring a continuous instruction supply to the backend.

The advent of cloud computing and the emergence of new server applications [6, 13, 14, 21, 38, 40, 48, 76], along with modern software practices emphasizing functionality, modularity, and portability [79], have led to the development of programs whose active instruction working sets overwhelm reasonably sized on-chip L1is/iTLBs/BTBs. Failing to find instructions and closely related information such as iTLB and BTB entries to efficiently fetch instructions causes fetch stalls, resulting in pipeline bubbles and reduced instruction supply to the backend.

This problem necessitates the ability to move instructions and other closely related information required from the memory hierarchy (or secondary structures) to the primary structures (L1i/iTLB/BTB) before the processor references them. This would result in an improved instruction supply to the processor backend.

The last few decades have seen significant innovation in *instruction prefetching* techniques [9, 14, 29, 30, 34, 39, 41, 42, 43, 44, 45, 62, 64, 75] that seek to tackle the problem of instruction supply for applications with a large code footprint. These techniques primarily aim to preload the L1i with blocks of code ahead of time. Many of these techniques are closely integrated with the instruction fetching process. Some leverage the logic used in instruction fetching to run ahead in the instruction stream, allowing them to tolerate the latency involved in fetching instructions from a lower-level cache. Other techniques are triggered by specific events within the instruction fetch process to preload blocks into the L1i.

However, these techniques are inherently tied to the limitations of instruction fetching logic. For instance, they depend on the accuracy of branch direction predictors and the Branch Target Buffer (BTB), making them vulnerable to branch mispredictions and BTB misses. Additionally, these methods necessitate separate techniques to manage other critical structures involved in instruction supply, such as the BTB [16, 18, 17] and the instruction Translation Lookaside Buffer (iTLB).

The actual instruction fetch process must accurately get past all the control instructions to identify the exact sequence of instructions to execute. However, the problem we are addressing is how to supply the L1i, iTLB, and BTB with the necessary blocks of instructions, iTLB entries, and BTB entries. This process can tolerate some imprecision, making the determination of a precise instruction stream less critical for this particular problem.

We pose the question: *Can we create a representation of the program that remains largely stable and facilitates resolving upcoming high-level control flow while determining the blocks, iTLB entries, and BTB entries that the processor is likely to reference?* To address this, we turn to high-level control flow, specifically control flow at the call graph level.

We observe that control flow at the call graph level is straightforward to capture and remains relatively stable, as has also been observed by others [9]. This stability facilitates the simple resolution of upcoming control flow without the need to execute the actual instructions in the program. Furthermore, it enables an operation separate from instruction fetch, allowing it to go past multiple program control instructions in each step.

This dissertation presents a high-level sequencing mechanism that can sequence the program at the callgraph level with near precision in many cases, accurately identifying where the processor is likely to be and determining the blocks of instructions, iTLB entries, and BTB entries needed by the processor. This unified scheme is used to supply the L1i, iTLB, and BTB in time for processing the requisite instructions. This scheme constructs a shadow program representation that remains largely unchanged to facilitate this process and does not require any execution resources. Furthermore, this scheme operates independently of the logic for fetching instructions. This technique uses identifiers validated by the processor to ensure it stays on track. The precise identification of the upcoming high-level control flow results in significant reductions in misses for all three key microarchitectural structures, often approaching the performance of a perfect L1i, iTLB, and

BTB, especially in cases where there are frequent misses in these structures with more conventional approaches.

1.1 Contributions

This dissertation makes the following key contributions:

High-Level Control Flow Characteristics: We present observations to show high-level control flow at the callgraph level is relatively stable and easy to capture in practical-sized hardware tables for these applications. This can be leveraged to run ahead in the instruction stream, identifying the blocks of instructions the processor is likely to need.

Instruction Presending: Building on these observations, we present a microarchitectural technique ***Instruction Presending*** which can determine where the processor is likely to be at a high-level and use this information to identify and proactively move blocks of instructions that the processor is likely to need, independent of the logic used by the processor to fetch instructions.

Presending for an iTLB/BTB: We present enhancements to Instruction Presending that can be used to move instruction TLB and BTB entries from secondary structures to primary structures (iTLB/BTB).

1.2 Dissertation Outline

This dissertation is organized as follows:

Chapter 2 provides a detailed description and analysis of the benchmarks used in this work. Additionally, it includes a comprehensive description of the experimental infrastructure employed in this study.

Chapter 3 provides a detailed description of the prior work in the area and presents a taxonomy of schemes used to prefetch blocks of instructions, BTB entries, and iTLB entries. It explores the potential for a unified scheme based on empirical observations and explains how it differs from existing techniques that address the instruction supply problem. We observe that control flow at the call graph level is relatively stable and can be maintained in reasonably sized tables. Furthermore, we discuss some benefits of sequencing high-level control flow. This leads to the key technique of Instruction Presending, which we present in the next chapter.

Chapter 4 presents a comprehensive description of the Instruction Presending technique. We provide a description of the different structures used by the technique, describe how they are constructed and how it operates to traverse high-level control flow, identifying and moving blocks of instructions likely to be referenced by the processor.

Chapter 5 presents a comprehensive evaluation of the Instruction Presending technique. Here we evaluate the efficacy of the high-level sequencing, miss reductions in the L1i and the reduced time waiting for instructions with Instruction Presending. Further, we also evaluate different parts of the design trying to help understand where the benefits are coming from. We observe that Instruction Presending achieves a very high accuracy in sequencing control flow at the call graph level, resulting in performance close to that of a perfect L1i in many cases.

Chapter 6 details the enhancements necessary to presend iTLB entries and provides an evaluation of these enhancements, with a focus on miss reductions in the iTLB. Similar to the L1i, Instruction Presending results in performance close to that of a perfect iTLB in many cases, with these enhancements.

Chapter 7 details the enhancements necessary to presend BTB entries and provides an evaluation of these enhancements, with a focus on miss reductions in the BTBs. Similar to the L1i and the iTLB, Instruction Presending results in performance close to that of a perfect BTB in many cases, with these enhancements.

Chapter 8 presents concluding remarks, outlines directions for future work and present some reflections.

Chapter 2

Benchmarks and Experimental Infrastructure

In this chapter, we provide a detailed description of the benchmarks used in this dissertation. Following that, we present data to show the code referencing characteristics of these benchmarks, which helps motivate our study of these benchmarks for this work. Towards the end, we also provide a description of the simulation infrastructure used.

2.1 Benchmark Description

For this work, we utilized 100 server benchmark traces provided by Qualcomm Datacenter Technologies, released following the Championship Value Prediction (CVP) [2]. These traces enable the measurement of various statistics related to different parts of the CPU pipeline. Additionally, they include register and memory dependencies, allowing for the simulation of an out-of-order core and the collection of performance metrics such as instructions per cycle. Unlike traces obtained with Pin [50], these traces encompass system activity. However, the traces are anonymized, meaning the actual workload is unknown, and some information, such as addressing mode and exact opcodes, has been removed. Qualcomm used these traces to evaluate their CPUs for servers. These benchmark traces have been employed in multiple papers on prefetching [11, 22, 63, 64] and in the Value Prediction Championship [2] due to the large number of traces and their industry origin.

2.2 Data Presentation

For all plots presented, the x-axis represents the various benchmark programs (encompassing all 100 benchmarks), while the y-axis represents the metric of interest.

Next, we present some of the characteristics of these benchmarks.

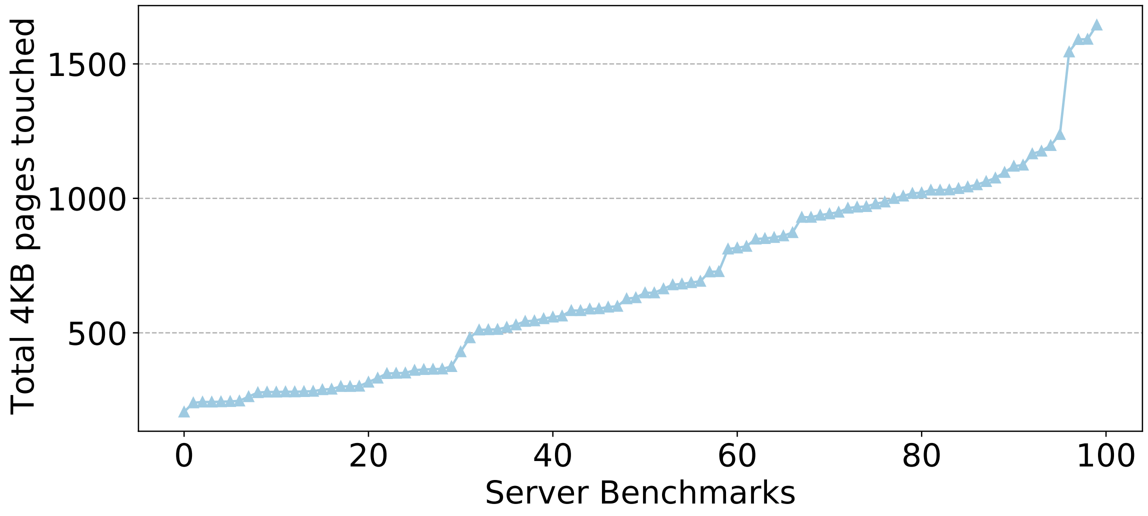


Figure 2.1: Code Pages References

2.3 Code Footprint

Figure 2.1 presents the total number of 4KB code pages touched by all benchmarks. The code footprint ranges from 0.8MB to 6.5MB, leading to frequent misses in a reasonably sized L1 instruction cache (L1i), instruction TLB (iTLB), and branch target buffer (BTB), which we quantify next.

2.3.1 L1i MPKI and Miss Rates

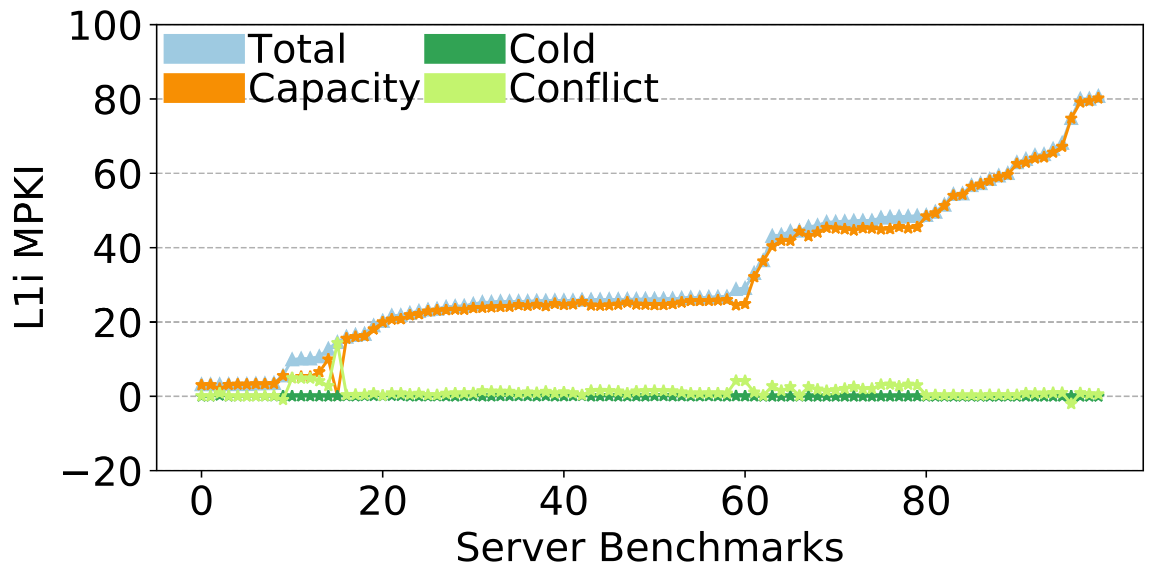


Figure 2.2: L1i (32KB) MPKI Breakdown

Figure 2.2 presents the L1i *Misses Per Kilo Instructions* (*MPKI*) for a 32KB L1i with an associativity of 8 and a cache block size of 64B. We observe frequent misses, with L1i MPKI ranging from 3 to 81. The figure also breaks down the misses into cold, capacity, and conflict misses for all benchmarks [33]. The contribution from cold misses is close to 0 for all benchmarks. Conflict misses account for a large fraction of misses in benchmarks with smaller L1i MPKIs. In a few cases, the conflict misses are negative (or we have anti-conflict misses) because using a fully associative cache results in more misses compared to the set associative cache. This is a well-known result when there is a "looping" access pattern [33, 71]. For most benchmarks, especially those with larger L1i MPKIs, the majority of misses are capacity misses. Given that most benchmarks have L1i MPKIs dominated by capacity misses, we next study the L1i MPKI for larger cache size.

Figure 2.3 presents the L1i MPKI, with the y-axis on a logarithmic scale, for 32KB, 64KB, and 128KB L1i caches across all benchmarks. The associativity is 8

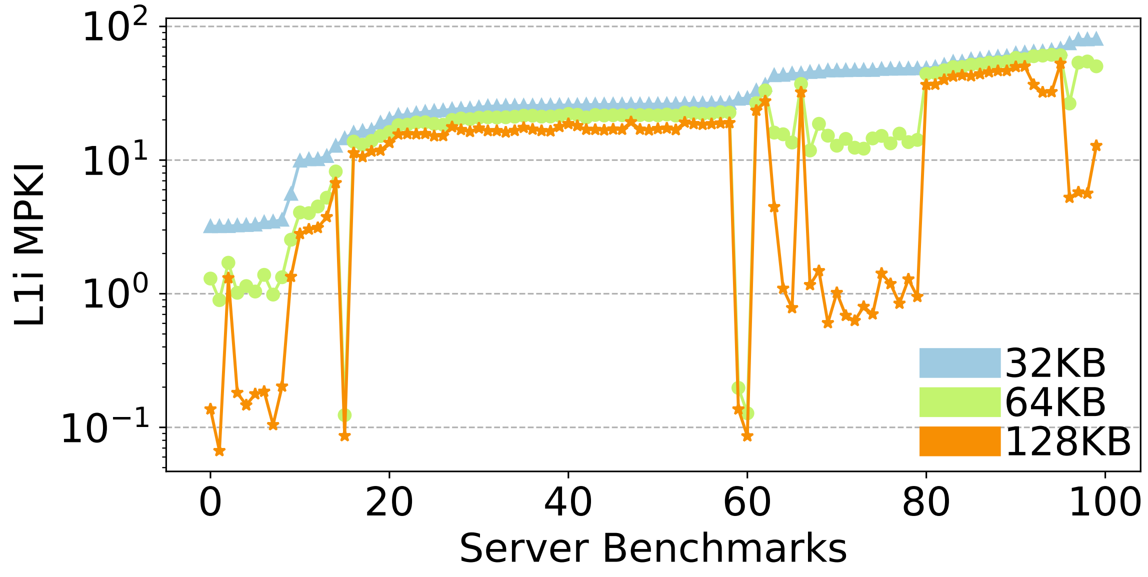


Figure 2.3: L1i MPKI

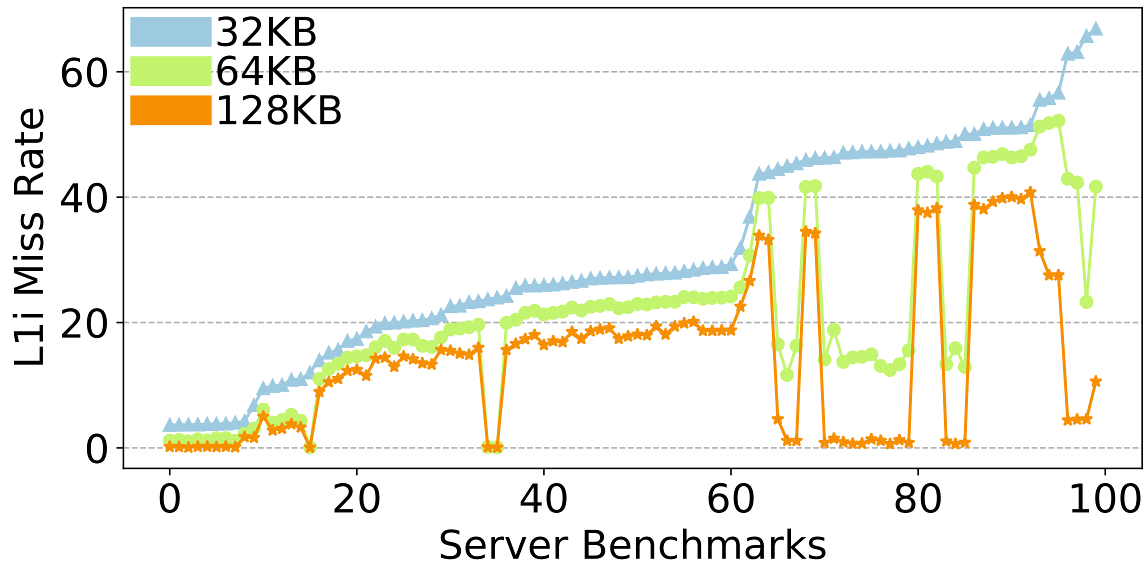


Figure 2.4: L1i Miss Rate

in all cases, and the cache block size is 64B. Many applications also experience frequent misses with 64KB and 128KB instruction caches, with L1i MPKI reaching as high as 62 and 53 for these cache sizes, respectively.

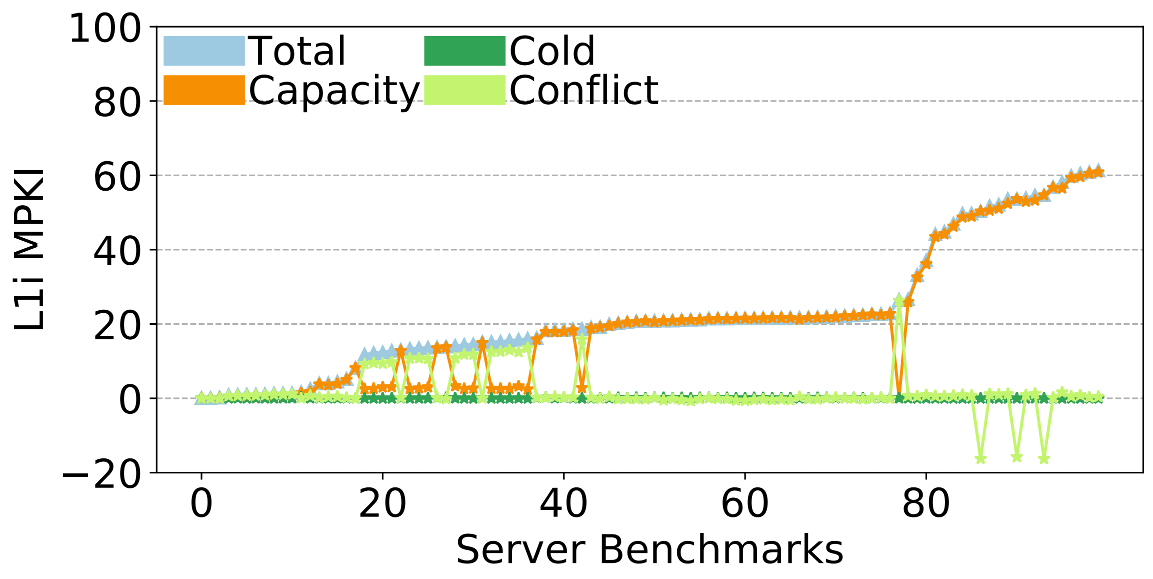


Figure 2.5: L1i (64KB) MPKI Breakdown

Figure 2.4 presents the L1i miss rates for 32KB, 64KB, and 128KB L1i caches across all benchmarks. The associativity is 8 in all cases, and the cache block size is 64B. The L1i miss rates range from 3.5% to 67% for a 32KB instruction cache. Many applications also experience frequent misses with 64KB and 128KB instruction caches, with L1i miss rates reaching as high as 52.2% and 41% for the two cache sizes, respectively.

Finally, Figures 2.5 and 2.6 present the breakdown of misses into cold, capacity, and conflict misses for both 64KB and 128KB caches, with an associativity of 8. We observe that many benchmarks with non-trivial L1i MPKI remain dominated by capacity misses, even with these larger L1is. Similar to what we saw earlier, the negative conflict misses arise when a fully associative cache results in more misses compared to a set associative cache.

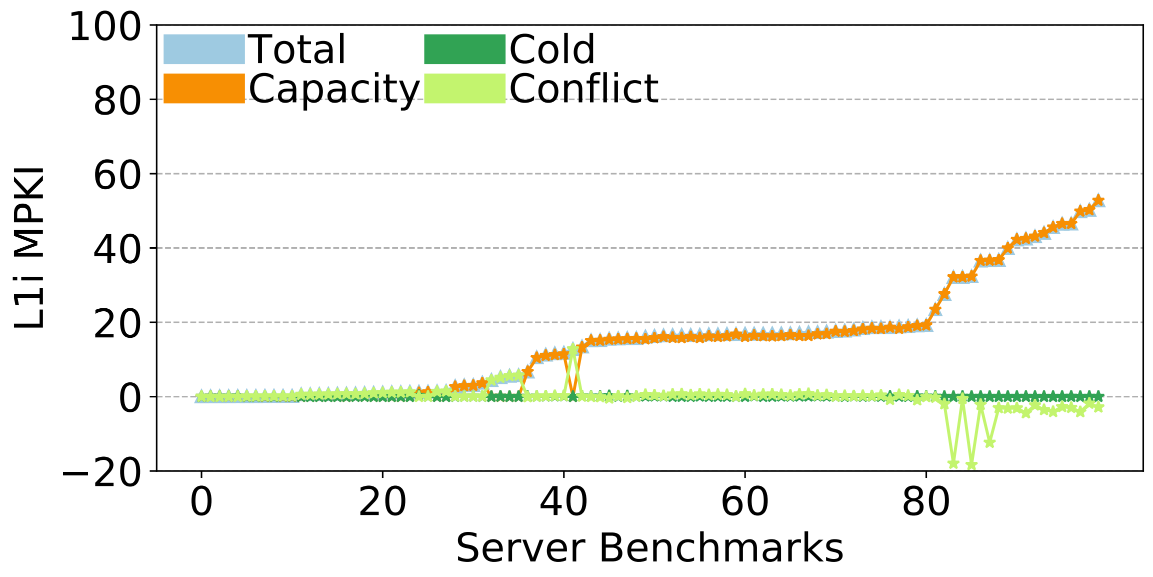


Figure 2.6: L1i (128KB) MPKI Breakdown

All these plots indicate frequent code movement from a lower-level cache to the L1i for benchmarks with a large code footprint.

2.3.2 iTLB MPKI (64-entry and 128-entry)

Figure 2.7 presents the instruction TLB (iTLB) MPKI for a 64-entry iTLB with an associativity of 4 and a base page size of 4KB. We observe frequent misses, with iTLB MPKI ranging from 0.5 to 13. The figure also breaks down the misses into cold, capacity, and conflict misses for all benchmarks. The contribution from cold misses is close to 0 for all benchmarks. Conflict misses account for a large fraction of misses in a few benchmarks. In a few cases, the conflict misses are negative, similar to what we saw for the L1i. For most benchmarks, especially those with larger iTLB MPKIs, the majority of misses are capacity misses. Given that most benchmarks have iTLB MPKIs dominated by capacity misses, we next study the iTLB MPKI for a larger iTLB.

Figure 2.8 presents the iTLB misses per kilo instructions (MPKI), with the y-

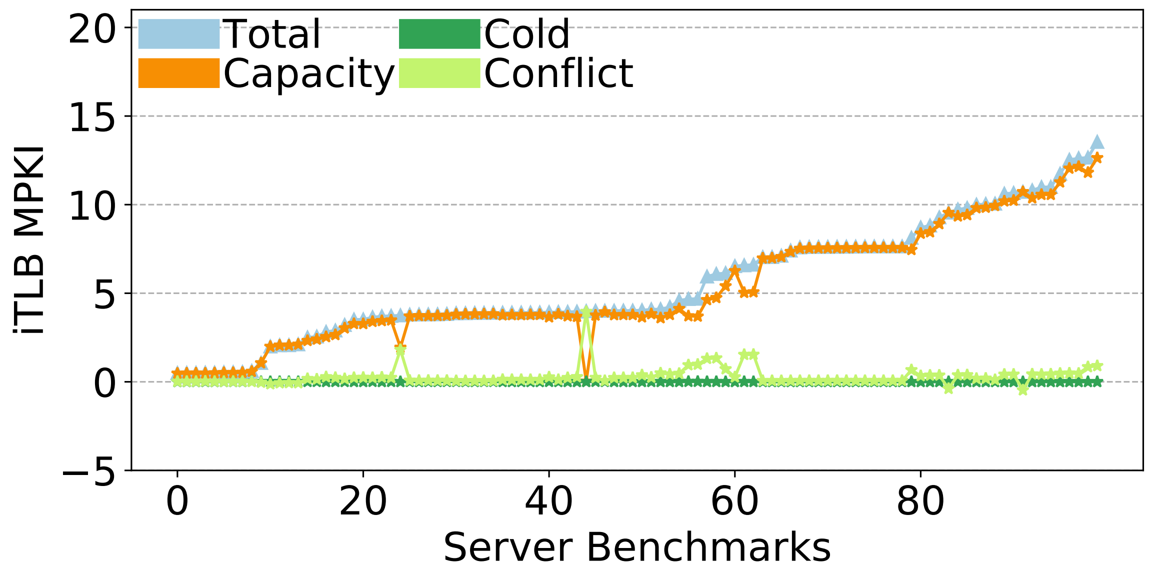


Figure 2.7: iTLB (64-entry) MPKI Breakdown

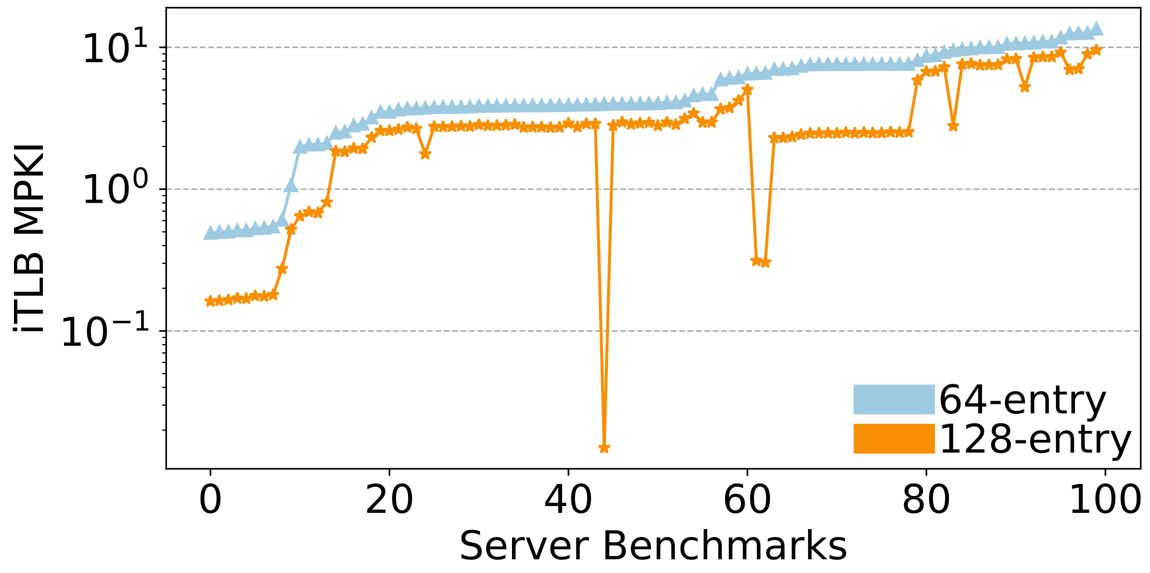


Figure 2.8: iTLB MPKI

axis on a logarithmic scale, for both a 64-entry iTLB and a 128-entry instruction TLB across all benchmarks.Many applications also experience frequent misses even with a 128-entry instruction TLB.

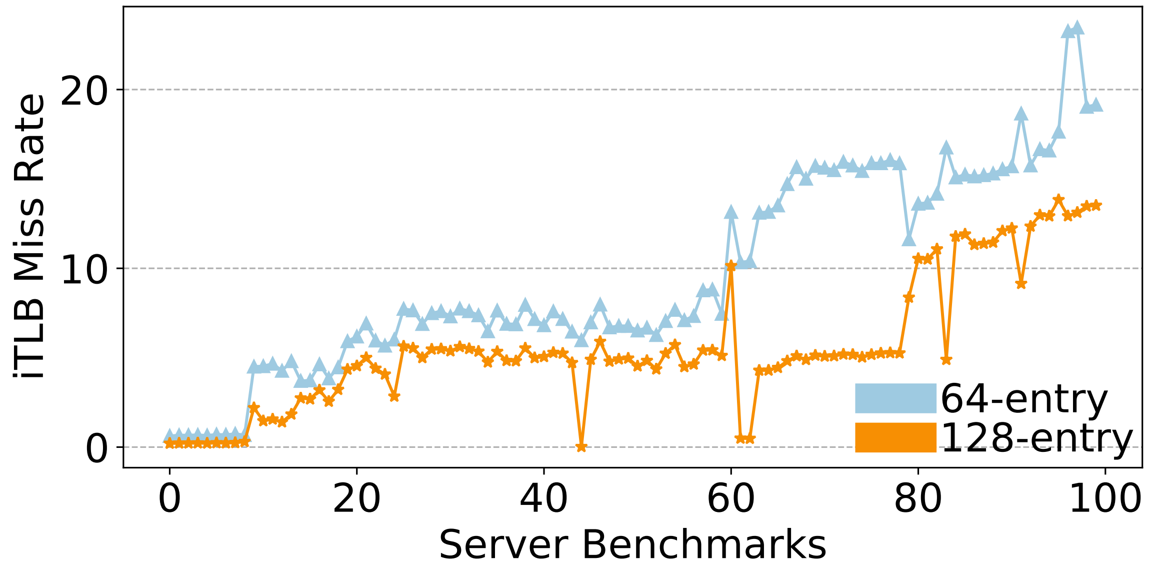


Figure 2.9: iTLB Miss Rate

Figure 2.9 presents the iTLB miss rates for both a 64-entry iTLB and a 128-entry iTLB across all benchmarks. Miss rates range from 0.6% to 23%, for a 64-entry iTLB. Many applications also have high miss rates with a 128-entry instruction TLB, with miss rates as high as 14%.

Figure 2.10 presents a breakdown of misses into cold, capacity, and conflict with a 128-entry TLB and an associativity of 4. Many benchmarks with a non-trivial iTLB MPKIs continue to be dominated by capacity misses.

2.3.3 BTB MPKI

Figure 2.11 presents the BTB MPKI for a 512-entry BTB with an associativity of 8. We observe frequent misses, with BTB MPKI ranging from 2 to 60. The figure also breaks down the misses into cold, capacity, and conflict misses for all bench-

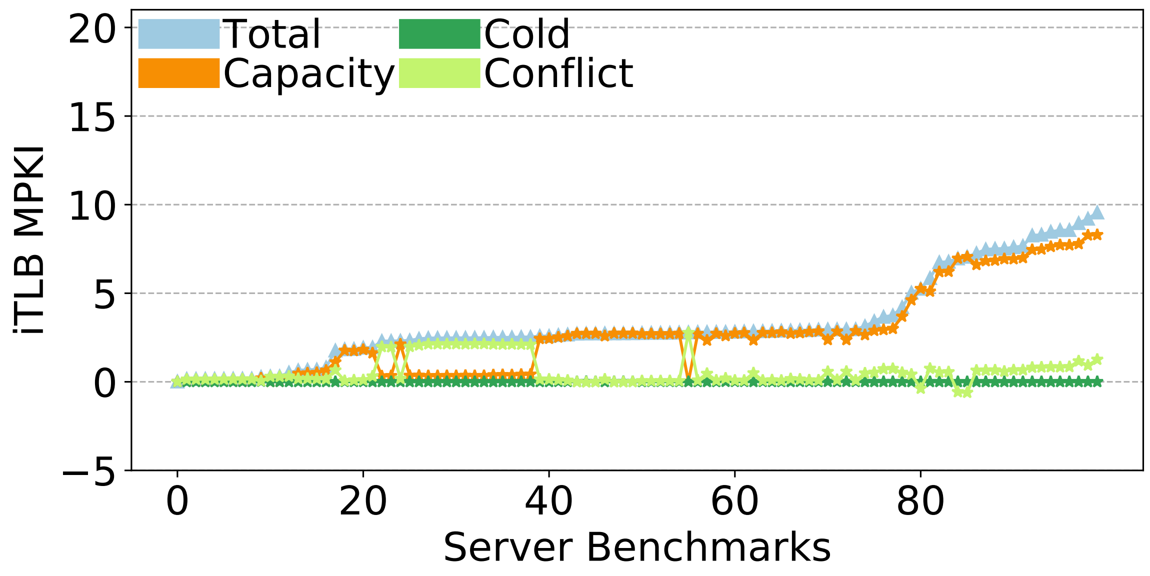


Figure 2.10: iTLB (128-entry) MPKI Breakdown

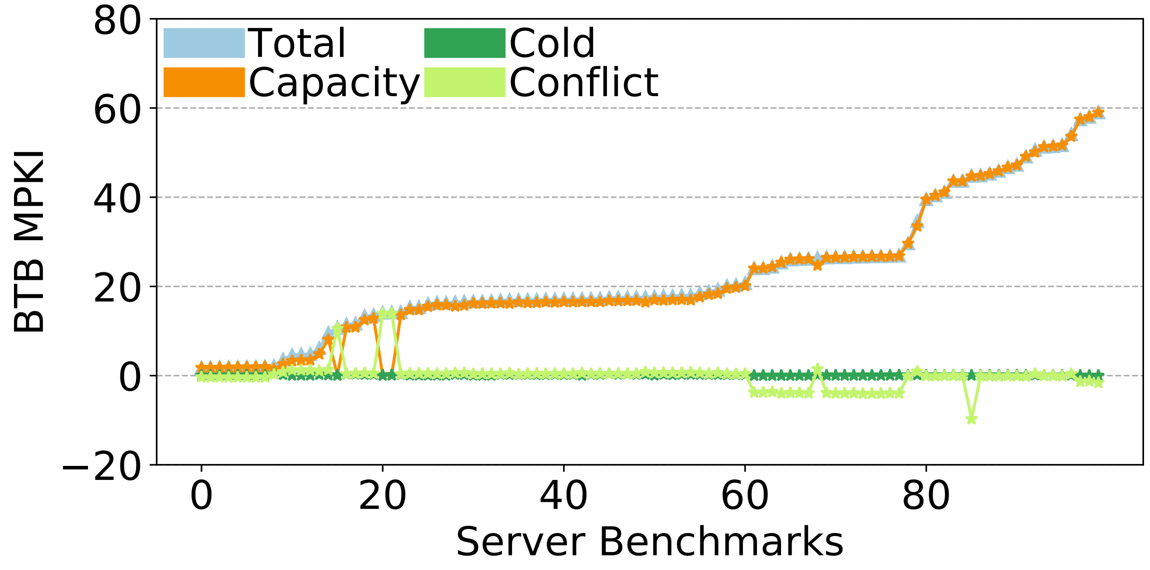


Figure 2.11: BTB (512-entry) MPKI Breakdown

marks. The contribution from cold misses is close to 0 for all benchmarks. Conflict misses account for a large fraction of misses in a few benchmarks. For some of the benchmarks, the conflict misses are negative, similar to what we saw for the L1i and iTLB. For most benchmarks, especially those with larger BTB MPKIs, the majority of misses are capacity misses. Given that most benchmarks have BTB MPKIs dominated by capacity misses, we next study the BTB MPKI a for larger BTB.

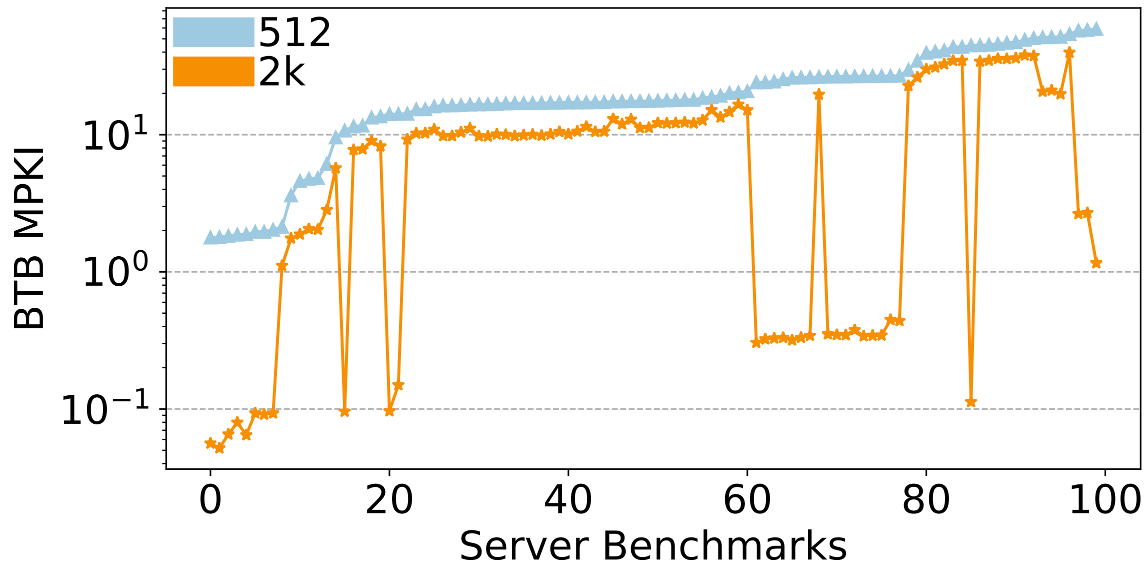


Figure 2.12: BTB MPKI

Figure 2.12 presents the branch target buffer (BTB) misses per kilo instructions (MPKI), with the y-axis on a logarithmic scale, for a 512-entry BTB, 1K-entry BTB and a 2K-entry BTB across all benchmarks. The associativity of the BTB in all cases is 8. Many applications also experience frequent misses with a 1K-entry and a 2K-entry BTB.

Figure 2.13 presents the BTB miss rates for a 512-entry, 1K-entry, and a 2K-entry BTB across all benchmarks. Miss rates range from 1% to 41%, for a 512-entry BTB. Many applications also have high miss rates with a 1K-entry BTB and a 2K-entry BTB, with miss rates as high as 32% and 26% respectively.

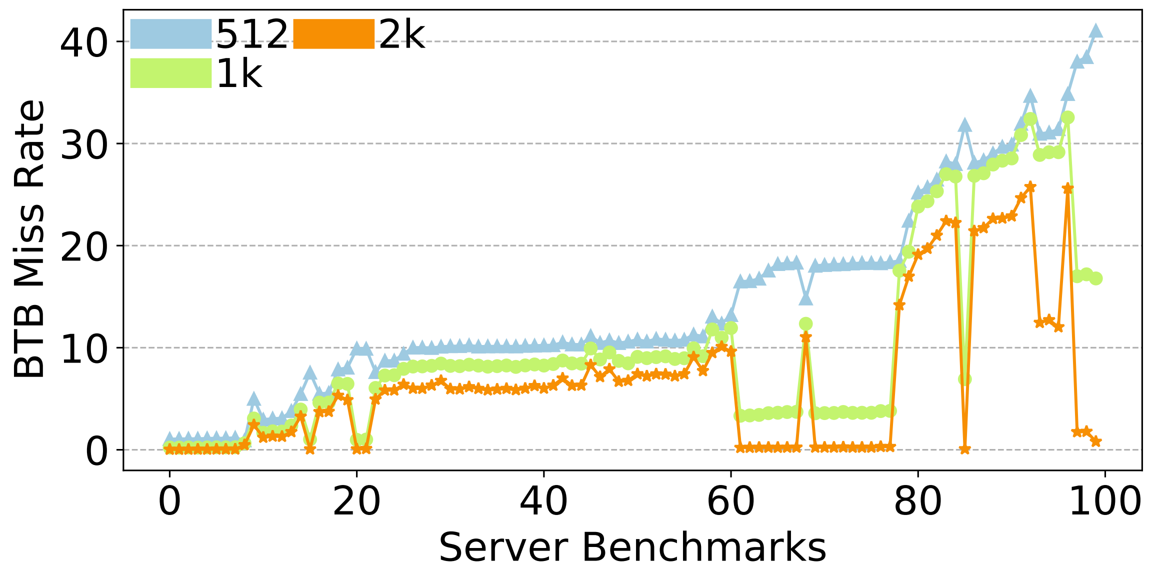


Figure 2.13: BTB Miss Rate

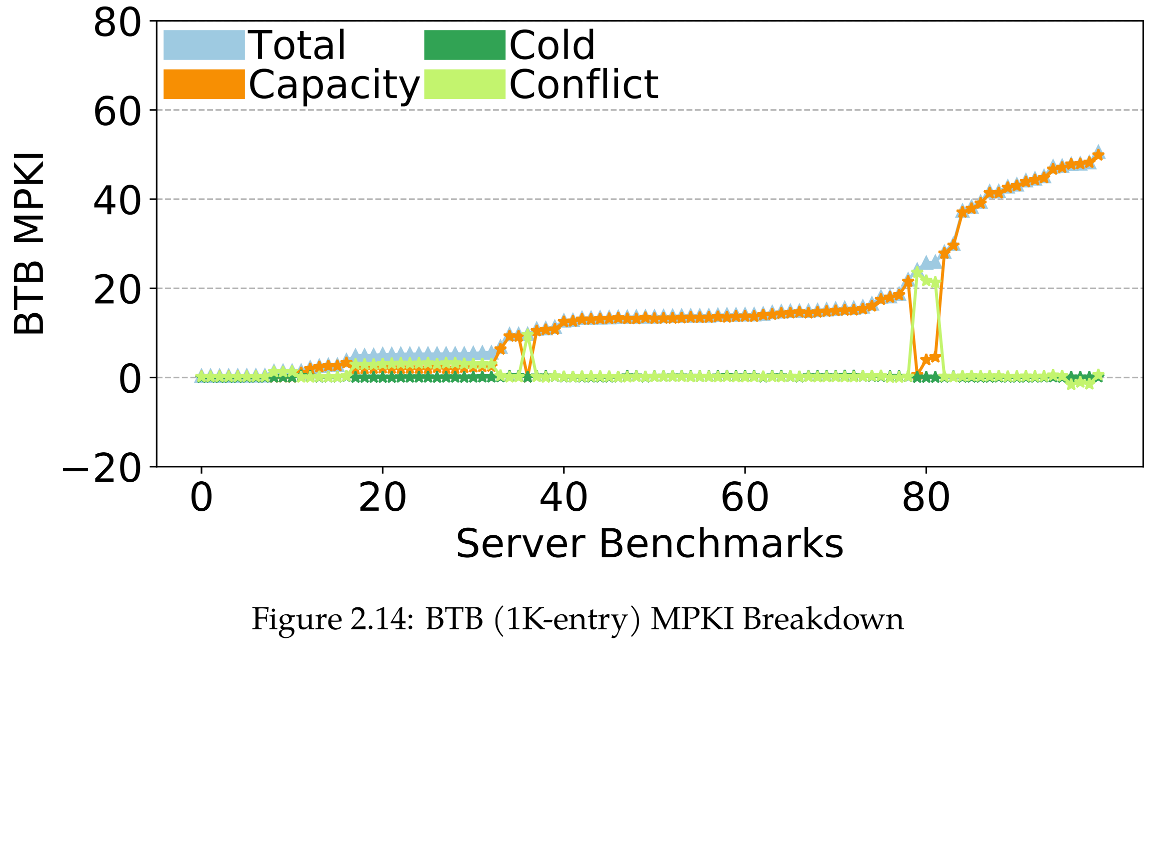


Figure 2.14: BTB (1K-entry) MPKI Breakdown

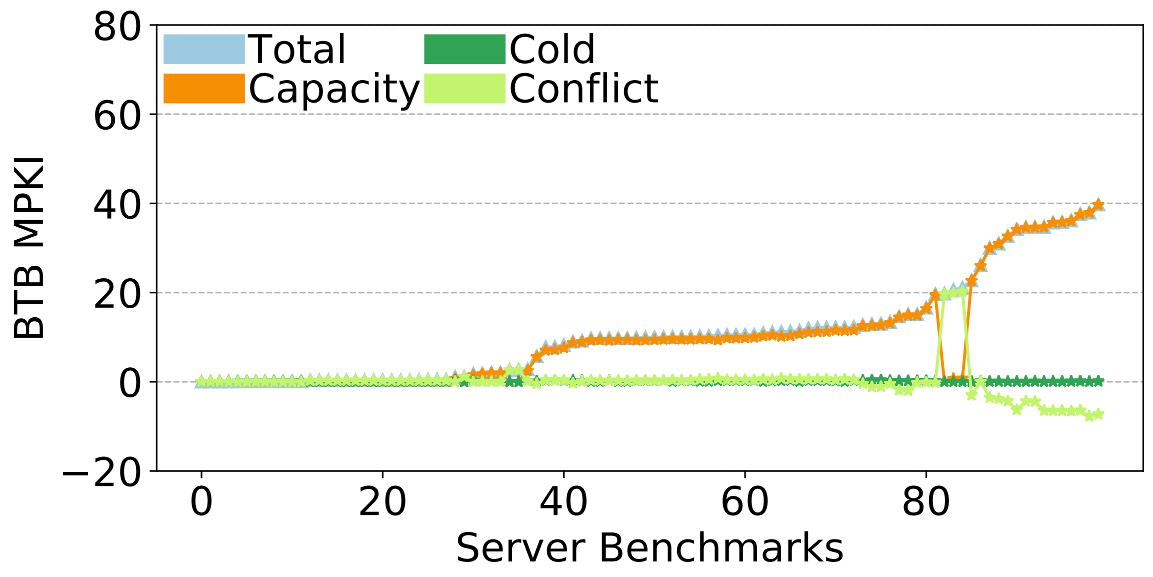


Figure 2.15: BTB (2K-entry) MPKI Breakdown

Finally, Figures 2.14 and 2.15 present the breakdown of misses into cold, capacity, and conflict misses for both 1K-entry and 2K-entry BTB, with an associativity of 8. We observe that many benchmarks with non-trivial BTB MPKI remain dominated by capacity misses, even with these larger BTB sizes.

2.4 Why study these Benchmarks?

These applications exhibit a large code footprint and experience frequent misses in the key structures involved in the instruction fetch process, namely the iTLB, BTB and L1i. Further, these misses persist even with relatively larger sized structures. Lastly, these applications have been used in multiple papers on instruction prefetching [11, 22, 63, 64] to study the instruction supply problem. All these reasons collectively make these applications well-suited for studying the instruction supply problem, which is the main focus of this dissertation.

2.5 Tabular Unified Presentation

While we presented all the data as plots to showcase trends more effectively, we now present these metrics for all benchmarks in a table to provide the reader with a comprehensive understanding of all benchmarks studied in this dissertation, as shown in Table 2.1. The first three data columns quantify the L1i MPKI with a 32KB, 64KB, and 128KB L1i (all 8-way associative). The fourth and fifth data columns quantify the iTLB MPKI with a 64-entry and a 128-entry iTLB (both 4-way associative). The next three data columns quantify the BTB MPKI with 512, 1024, and 2048 entries (all 8-way associative).

	MPKI							
		L1i		**iTLB**			**BTB**	
App	**32K**	**64K**	**128K**	**64**	**128**	**512**	**1K**	**2K**
secret121	3.2	1.3	0.1	0.5	0.2	1.8	0.4	0.1
secret141	3.2	0.9	0.1	0.5	0.2	1.8	0.4	0.1
public9	3.2	1.7	1.3	0.6	0.3	2.1	1.5	1.1
secret100	3.2	1.0	0.2	0.5	0.2	1.8	0.4	0.1
secret103	3.3	1.1	0.1	0.5	0.2	1.9	0.4	0.1
public27	3.3	1.0	0.2	0.5	0.2	1.9	0.4	0.1
public30	3.4	1.4	0.2	0.5	0.2	2.0	0.4	0.1
public31	3.5	1.0	0.1	0.5	0.2	2.0	0.4	0.1
public29	3.6	1.3	0.2	0.5	0.2	2.0	0.4	0.1
public40	5.6	2.5	1.3	1.1	0.5	3.6	2.2	1.8
server001	9.9	4.1	2.8	2.0	0.6	4.6	2.6	1.9
secret132	10.1	4.0	3.0	2.1	0.7	4.8	2.8	2.0
secret137	10.2	4.5	3.1	2.1	0.7	4.8	2.8	2.1
public8	10.7	5.2	3.8	2.1	0.8	6.1	3.8	2.8
public57	12.8	8.2	6.7	3.7	1.8	9.6	6.9	5.7
server002	14.6	0.1	0.1	4.0	0.0	10.7	1.4	0.1
public11	16.0	13.7	11.3	2.5	1.9	11.5	9.7	7.8
public54	16.5	13.0	10.6	2.8	1.9	13.3	11.0	9.0
public1	16.8	13.9	11.6	2.6	1.8	11.7	9.7	7.8
server003	19.0	15.1	11.8	2.9	1.9	13.6	10.9	8.2
server004	20.3	16.2	13.5	3.2	2.3	14.2	11.3	9.2
public48	21.7	18.3	15.6	4.7	3.0	17.5	15.6	13.0
public49	21.8	18.4	15.7	4.7	3.0	17.5	15.6	12.9
secret128	22.4	19.1	15.7	3.5	2.6	15.4	12.8	10.3
server009	23.0	19.2	15.7	3.5	2.6	15.3	12.8	10.2
public55	23.3	18.7	15.1	4.0	2.8	18.5	15.4	12.8
public41	23.6	18.4	15.2	4.1	2.9	19.3	15.9	13.4
public51	24.0	20.0	17.7	4.2	3.1	18.8	16.6	15.1

Table 2.1 continued from previous page

	MPKI							
		L1i		**iTLB**			**BTB**	
App	**32K**	**64K**	**128K**	**64**	**128**	**512**	**1K**	**2K**
server010	24.2	20.5	16.9	3.7	2.7	16.1	13.5	10.9
public68	24.3	20.3	16.3	3.6	2.6	16.4	13.6	10.4
server011	24.8	20.8	17.3	3.7	2.7	16.6	13.9	11.1
secret111	25.3	20.9	16.5	3.9	2.8	16.4	13.3	9.8
secret131	25.3	20.8	16.6	3.9	2.8	16.3	13.3	9.8
public67	25.5	20.9	16.1	3.9	2.8	17.0	13.6	9.8
public6	25.6	21.1	16.6	3.9	2.8	16.9	13.8	10.0
public7	25.6	21.5	17.5	4.0	3.0	17.5	14.6	11.2
public3	25.6	21.5	17.0	3.8	2.8	17.1	14.0	10.5
public65	25.7	21.1	16.6	3.9	2.7	17.0	13.6	9.9
public66	25.7	21.2	16.5	3.9	2.9	17.0	13.8	10.0
server012	25.8	21.5	17.8	4.0	2.9	17.2	14.3	11.4
public46	25.8	22.1	18.7	5.9	3.7	20.1	18.2	14.7
public10	25.8	21.8	18.1	3.9	2.9	17.5	14.8	11.9
public69	26.0	20.8	16.9	4.0	2.8	17.6	14.1	11.3
secret105	26.0	21.7	16.9	3.9	2.7	16.7	13.4	9.7
public4	26.0	21.6	16.8	3.9	2.7	17.0	13.7	9.9
public2	26.1	21.6	17.0	3.8	2.8	17.2	14.1	10.5
secret120	26.1	21.7	16.9	3.9	2.7	16.6	13.5	9.8
public50	26.1	21.7	19.3	4.6	3.4	20.4	18.1	16.6
public64	26.1	21.7	17.0	3.9	2.7	17.1	13.9	10.1
secret133	26.2	21.7	16.8	3.8	2.7	16.7	13.6	10.1
public62	26.2	21.6	17.1	3.8	2.8	17.2	14.1	10.6
public63	26.2	21.9	17.3	3.8	2.8	17.2	14.1	10.6
public5	26.3	21.5	16.8	3.9	2.7	17.2	13.9	10.1
public47	26.4	22.7	19.2	6.1	3.7	20.7	18.7	15.1
public0	26.6	22.4	18.7	3.9	2.9	17.8	15.1	12.1

Table 2.1 continued from previous page

	MPKI							
	L1i			iTLB		BTB		
App	**32K**	**64K**	**128K**	**64**	**128**	**512**	**1K**	**2K**
public61	26.6	22.2	18.4	4.0	2.9	18.0	14.9	12.1
secret12	26.7	22.3	18.6	4.1	3.0	17.8	14.9	12.2
public60	26.8	22.8	18.9	4.0	2.9	18.0	15.2	12.3
server013	26.8	22.7	19.0	4.0	2.9	17.7	14.9	12.2
server014	28.8	0.2	0.1	6.6	0.3	14.2	1.4	0.1
server015	29.1	0.1	0.1	6.6	0.3	14.1	1.4	0.1
public53	33.2	26.7	23.5	6.1	4.2	26.4	22.0	19.7
server016	36.6	33.2	27.6	6.5	5.0	29.7	28.1	22.7
server017	43.2	16.1	4.5	7.0	2.3	24.1	4.9	0.3
server018	43.4	15.6	1.1	7.0	2.3	24.1	4.9	0.3
server019	44.4	13.5	0.8	7.1	2.3	24.4	5.0	0.3
public52	44.6	37.1	32.2	8.1	5.9	34.6	29.9	26.2
public34	45.6	11.8	1.2	7.6	2.5	26.8	5.3	0.3
server020	46.0	18.7	1.5	7.4	2.4	25.4	5.2	0.3
public39	46.9	15.2	0.6	7.6	2.5	26.7	5.3	0.3
public76	46.9	12.8	1.0	7.6	2.5	26.7	5.4	0.4
public38	47.1	14.4	0.7	7.6	2.5	26.5	5.3	0.4
secret10	47.2	12.4	0.6	7.6	2.5	26.1	5.3	0.3
public36	47.3	12.2	0.8	7.6	2.5	26.5	5.3	0.3
server021	47.3	14.5	0.7	7.6	2.5	26.0	5.2	0.3
public75	48.1	15.2	1.4	7.6	2.5	26.6	5.3	0.3
public33	48.3	13.3	1.2	7.6	2.5	26.7	5.3	0.3
public37	48.3	15.8	0.8	7.6	2.5	26.8	5.6	0.4
server022	48.4	13.6	1.3	7.6	2.5	26.2	5.3	0.3
public35	48.6	14.2	0.9	7.6	2.5	26.9	5.6	0.4
server023	48.7	44.2	36.6	8.7	6.7	39.6	37.4	30.0
server024	49.7	44.9	36.8	8.8	6.8	40.4	38.3	31.0

Table 2.1 continued from previous page

	MPKI							
	L1i			**iTLB**		**BTB**		
App	**32K**	**64K**	**128K**	**64**	**128**	**512**	**1K**	**2K**
server025	51.6	47.0	39.9	9.3	7.3	41.2	39.4	32.7
server026	54.4	49.7	42.3	9.7	7.6	43.7	41.8	34.7
server027	54.6	49.7	43.1	9.8	7.7	43.7	41.7	34.6
server028	56.8	51.7	42.5	10.6	8.3	44.9	42.8	34.2
server029	57.3	52.1	44.1	10.7	8.3	45.3	43.3	34.7
server030	58.4	53.6	45.6	10.8	8.5	45.9	44.2	35.8
server031	59.4	53.9	46.5	11.0	8.6	46.7	44.6	35.7
secret113	60.0	54.6	46.6	11.0	8.6	47.3	45.1	36.2
server032	62.9	58.2	49.9	11.7	9.2	49.1	47.4	37.9
public44	63.8	56.9	50.3	12.7	9.0	50.6	47.3	37.6
server033	64.9	59.9	36.7	10.0	7.5	51.3	47.9	20.6
server034	65.1	60.5	32.2	10.0	7.5	51.4	48.2	21.1
server035	66.6	61.3	32.4	10.1	7.5	51.7	48.0	19.8
public45	68.2	60.8	52.8	13.5	9.6	54.1	50.6	39.8
server036	74.8	26.5	5.2	9.5	2.8	44.8	9.7	0.1
server037	80.0	53.6	5.8	12.5	7.0	57.5	25.7	2.6
server038	80.1	54.7	5.6	12.6	7.1	58.0	25.9	2.7
server039	80.8	50.3	12.8	10.7	5.3	59.0	24.1	1.2

Table 2.1: Cumulative Benchmark Characteristics

2.6 Simulation Infrastructure

For all simulations run to gather the results presented in this thesis, we have used the ChampSim simulator [1, 31].

ChampSim is a trace-driven simulation infrastructure widely used for studying similar problems, allowing easier replication of the work and facilitating comparisons with other studies conducted using ChampSim. Additionally, the commercial traces released by Qualcomm are ChampSim compatible, making ChampSim an apt choice for studying these applications.

ChampSim models a decoupled in-order front end and an out-of-order back end. It includes a detailed memory hierarchy with an L1i, L1 data cache, L2 cache, and a last-level cache, common to many general-purpose processors. It also models a TLB hierarchy and a variable-latency page walk. Trace files contain only virtual addresses, and ChampSim simulates arbitrary mappings from virtual to physical pages.

ChampSim features a configuration file that allows it to model various microarchitectures. This configuration file specifies many aspects of the CPU, such as fetch, decode, execution, and retire widths, as well as the latency for different components. The configuration file also allows for specifying the memory hierarchy configuration, including cache sizes, associativity, and latency.

For the microarchitectural technique presented in this thesis, modifications are made on top of the existing decoupled in-order front end. Most changes are implemented as part of a separate instruction prefetching module, with a few modifications to the cache and out-of-order (OOO) core modules.

Detailed simulator configurations are presented before the evaluation of both microarchitectural techniques.

2.7 Conclusions

This chapter presented the characteristics of the benchmarks studied in this dissertation. We provided data to quantify the code footprint and its impact on various microarchitectural structures, including the L1i, iTLB, and BTB. This was followed by a description of ChampSim, the simulation infrastructure used for all evaluations in this dissertation.

Chapter 3

Instruction Supply Problem

In this chapter, we first present a detailed treatment of handling non-sequential control flow, which we briefly introduced in Chapter 1. This aspect is crucial to effective instruction supply. By understanding how non-sequential control flow is managed, we can appreciate much of the prior work that is coupled with these components. Next, we quantify the magnitude of the large code footprint problem. Following this, we introduce a taxonomy for schemes that prefetch blocks of instructions into the L1i and describe many of these schemes. Similarly, we present a taxonomy of schemes to manage a BTB. We also briefly describe iTLB management schemes. Finally, we provide an empirical potential for a unified scheme that can supply many key structures involved in instruction supply, decoupled from the logic for fetching instructions.

3.1 Handling Program Non-Sequential Control Flow

Common program constructs such as loops, if-then-else, and procedure calling make control instructions inevitable in programs for controlling the flow of instructions. Control instructions redirect the flow of instructions during program execution, resulting in a non-sequential flow of instructions. Five main classes of control instructions result in non-sequential control flow, which we describe below.

- **Conditional Branches** - These instructions test a condition to decide whether the branch is fall-through or taken. It requires both the condition and the target address before the branch can take place. The taken target address is typically computed by adding an offset to the address of the branch instruction and the fall-through address is the next sequential address in the instruction stream.
- **Unconditional Direct Jumps and Direct Calls** - These instructions, as the name suggests unconditionally jump to the target address which is computed by adding an offset to the address of the jump or call instruction (similar to the taken address for a conditional branch).
- **Unconditional Indirect Jumps and Indirect Calls** - These instructions unconditionally jump to a target whose address is specified by a value generated by an earlier instruction that is typically stored in a register.
- **Returns** - These instructions jump to the return address of its corresponding subroutine call.
- **Traps** - These instructions unconditionally jump to the address of an operating system call handler. Further, these instructions require that all preceding instructions are complete before their execution, so no instructions following such instructions can execute speculatively. This mechanism ensures the system call is handled correctly and the program state remains consistent.

Control instructions give rise to a **Control Flow Graph (CFG)**[8, 55], where nodes represent sequential segments of the program, each constituting a **basic block** with a single entry and exit point. Edges in this graph signify control flow transfers. Program execution involves traversing this CFG, where the path taken is determined by the outcomes of control instructions, including conditions and targets of the control instructions.

A naive handling of control instructions indicates a problem - outcomes of conditional branches and targets of control instructions are not known immediately after the control instruction is fetched. This causes the stalling of the processor until the control instruction is decoded/executed which starts to affect the flow of instructions to the backend.

Modern processors incorporate specific mechanisms to address this issue and maintain a continuous flow of instructions to the backend. The fundamental approach involves leveraging run-time trained data structures designed to predict program control flow transfers by traversing the program's control flow graph.

- **Direction Predictors** [36, 52, 67, 70, 78, 84] - These tables utilize control flow history to predict the outcome of conditional branches, determining the path taken by the branch.
- **Branch Target Buffers (BTB)** [85] - These tables store the target addresses of previous instances of the control instructions, encompassing both conditional branches and unconditional jumps or calls. Entries in the BTB are identified by the addresses of the respective control instructions.
- **Return Address Stack (RAS)** [85] - This hardware stack is employed to efficiently retrieve the return address of a subroutine call for return instructions.

Figure 3.1 illustrates the logic for determining the next instruction (Program Counter or the PC) to fetch, which we explain next. When a conditional branch instruction is fetched, the Branch Target Buffer (BTB) is accessed to retrieve the target address, while the direction predictor is consulted to predict whether the branch will be taken or will fall through. For an unconditional jump, the direction predictor is not needed, and the BTB provides the target address directly upon fetching the instruction. In the case of a return instruction, the direction predictor is again not needed, and the Return Address Stack (RAS) is accessed to retrieve the target address. Following a non-control instruction, the fetching of instructions continues sequentially from the next instruction address (PC+4).

Putting it together, when a control instruction is fetched, the Branch Target Buffer (BTB), Direction Predictor, and Return Address Stack (RAS) are accessed using the instruction's address (and branch history for direction predictors). The

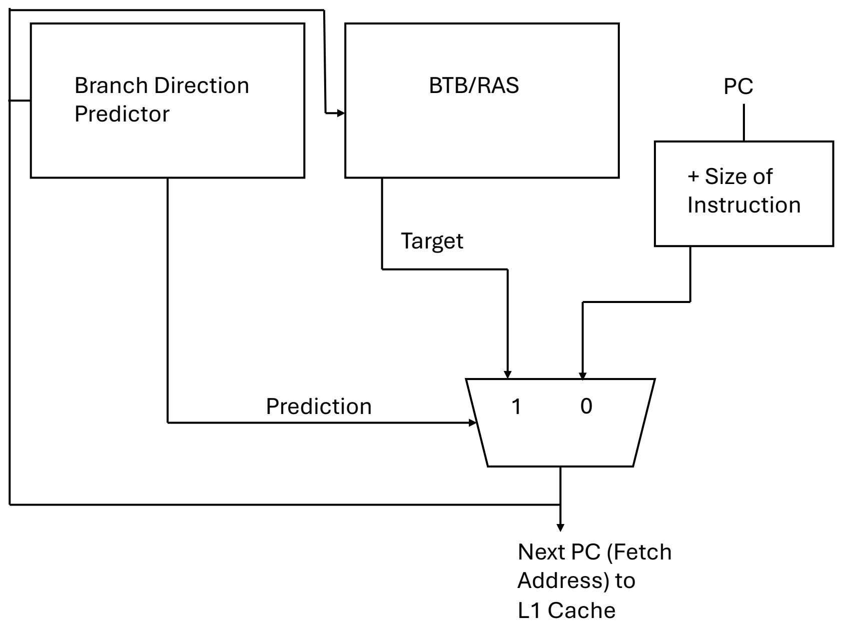

Branch Direction Predictor
BTB/RAS
PC
+ Size of Instruction
Target
Prediction
1
0
Next PC (Fetch Address) to L1 Cache

Figure 3.1: PC Determination Logic

target address retrieved from the BTB or RAS is then utilized to appropriately redirect the instruction flow in the subsequent cycle, ensuring a continuous stream of instructions to the backend of the machine. Achieving the correct flow of instructions requires high accuracy in all these components. This process constitutes the fundamental logic underlying the determination of the next Program Counter (PC).

3.2 Large Code Footprint Problem

The rise of cloud computing and the emergence of numerous new commercial applications [6, 13, 14, 21, 38, 40, 48, 76] have started to overwhelm the structures shown in Figure 3.1 (especially the BTB), as well as the L1i and the iTLB. Programs now often have active instruction working sets that exceed the capacity of these on-chip primary structures.

Larger instruction working sets result in frequent movement of instructions and related information between secondary and primary caches, leading to increased fetch latencies when instructions or iTLB entries are not found in the L1i or iTLB. These delays cause cycles where the processor cannot fetch instructions, resulting in pipeline bubbles. Similarly, a larger instruction working set leads to frequent BTB misses, creating pipeline bubbles due to the inability to fetch instructions past discontinuities. These pipeline bubbles significantly reduce instruction throughput and overall performance, which is particularly critical as processors fetch and issue multiple instructions per cycle. Studies indicate that instruction delays can contribute to 25-40% of the total execution time [32].

Next, we present some data to quantify the frequent movement for the commercial applications we study in this dissertation. We first describe the magnitude of the problem and then describe some implications.

3.2.1 Frequent Code Movement for the L1i

When an instruction fetch cannot be satisfied by the L1i, the request is handled by a lower-level cache (L2 cache) in the memory hierarchy. The frequency of cache misses is typically quantified using MPKI (Misses Per Kilo Instruction), which provides insight into the extent of data movement from lower-level to higher-level caches.

However, in the context of multi-issue processors where multiple instructions are fetched per cycle, and considering that processor implementations may buffer cache block contents until transitioning to fetch from another block, a more comprehensive metric is useful. Therefore, we introduce another metric known as *Misses Per Kilo Accesses* (*MPKA*). **MPKA** measures the misses per thousand L1i accesses and serves as a metric to understand how frequently a lower-level cache is involved in an L1i access. This metric helps to better quantify and comprehend the magnitude of the problem.

Figure 3.2 plots the L1i MPKI for a 32KB L1i, 6-wide fetch and 4-wide issue (note the log scale on the y-axis) for all the benchmarks considered. MPKIs vary

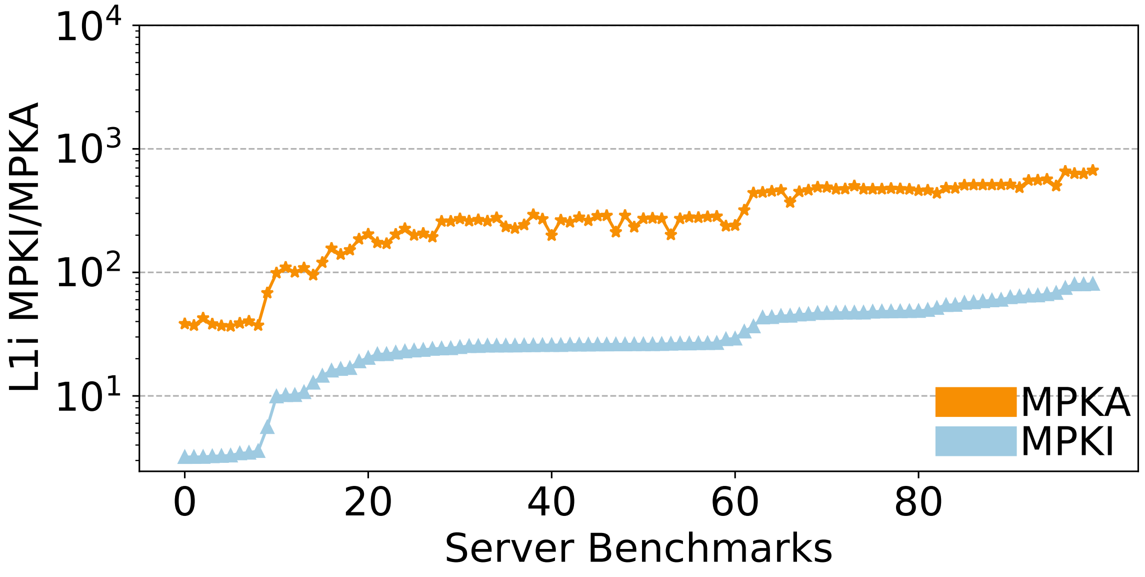


Figure 3.2: L1i MPKI/MPKA

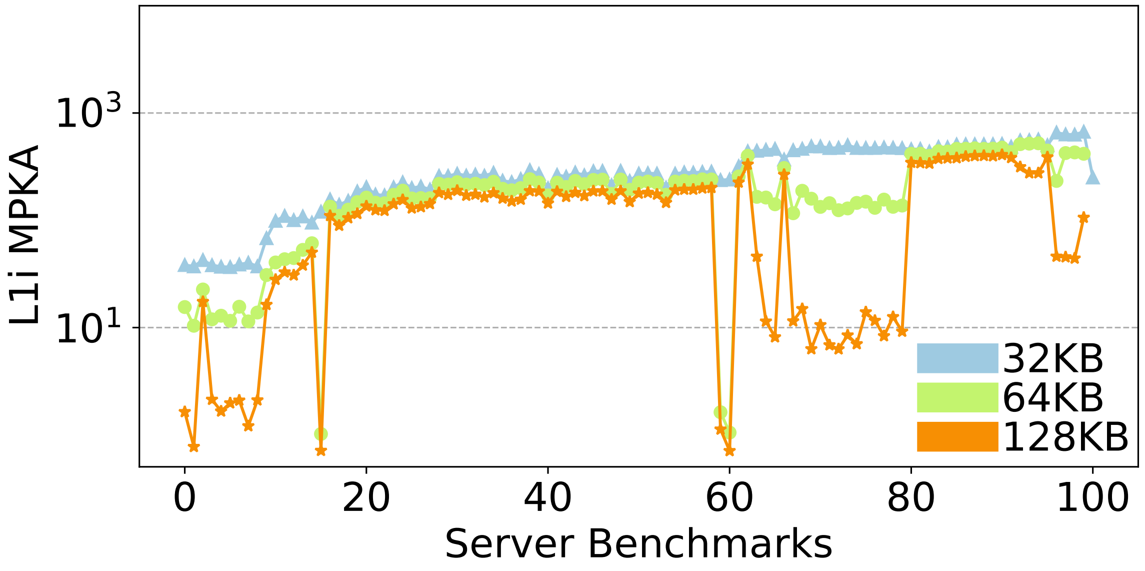


Figure 3.3: Varying L1i Size

from 3.2 to 80. L1i accesses used to quantify MPKA are counted based on effective block transitions or references to different blocks; consecutive accesses to the same block are combined into a single access. MPKAs are almost an order of magnitude higher for all benchmarks, because every access to the L1i fetches multiple instructions from a block before transitioning to fetch instructions from another block. The average MPKA for the benchmarks considered is 190, that is almost every 1 in 5 L1i accesses involves a lower-level cache access. For some benchmarks, the MPKAs are more than 500 (more than 1 in every 2 L1i accesses involve a lower-level cache). This data indicates that for the applications considered, there is a very frequent movement of cache blocks from the lower-level cache to the upper-level cache.

Figure 3.3 considers the L1i MPKA for these benchmarks for a 32KB and a 128KB L1i. We observe that even with a 64KB L1i and a 128KB L1i, for many of the benchmarks considered, the larger MPKA persists, continuing to frequently involve a lower-level cache.

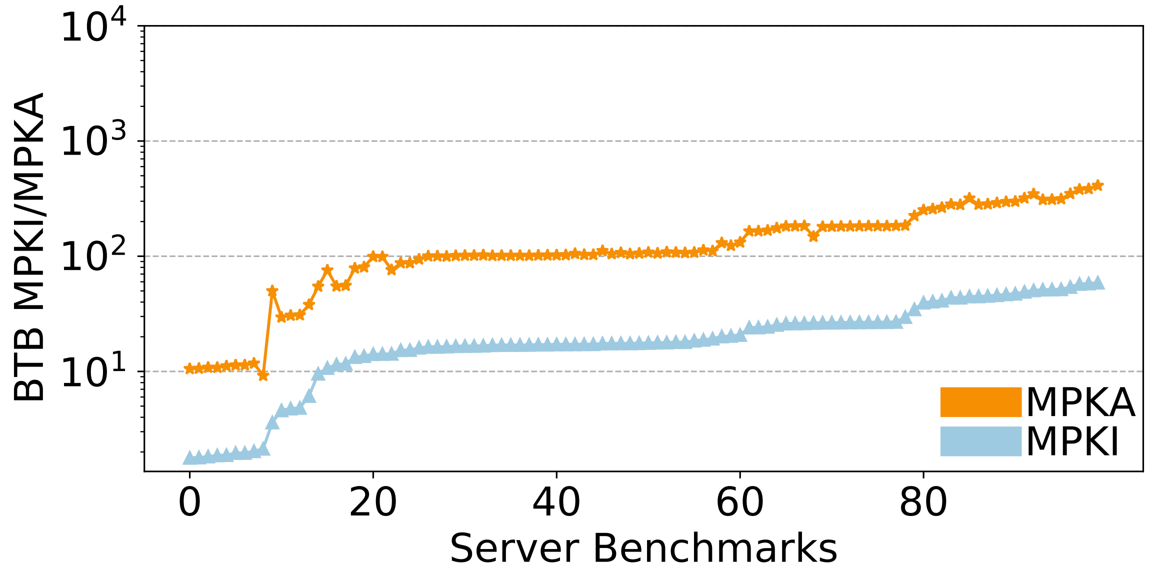


Figure 3.4: BTB MPKI/MPKA

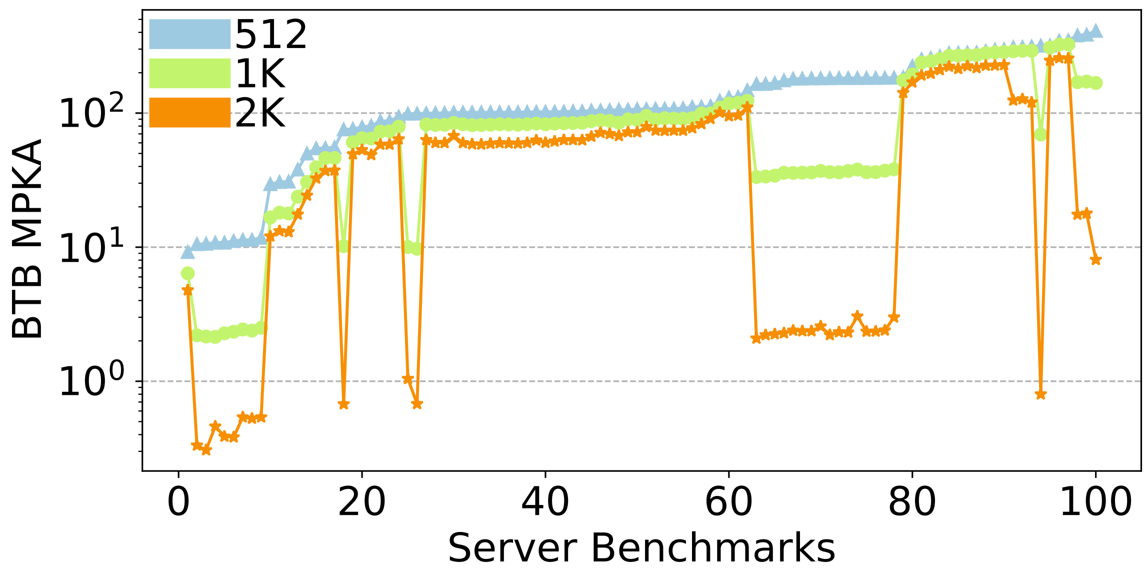


Figure 3.5: Varying BTB Size

3.2.2 Frequent Misses for the L1 BTB

Figure 3.4 examines BTB misses for a 512-entry BTB (note the log scale on the y-axis), which is a reasonably sized L1 BTB. We observe that a 512-entry BTB experiences frequent misses for many of the applications considered, with the BTB MPKI ranging from 1 to 59. Similar to the L1i analysis, we also quantify the frequency of BTB misses as a fraction of BTB accesses using the MPKA metric. BTB MPKA is as high as 300-400 for some of the benchmarks considered, indicating that as high as 1 in 2-3 BTB accesses results in a miss.

Figure 3.5 considers the BTB MPKA for these benchmarks for a 512-entry, 1K-entry and a 2K-entry BTB. We observe that even with bigger BTBs, for many of the benchmarks considered, the larger MPKA persists, continuing to frequently result in fetch stalls.

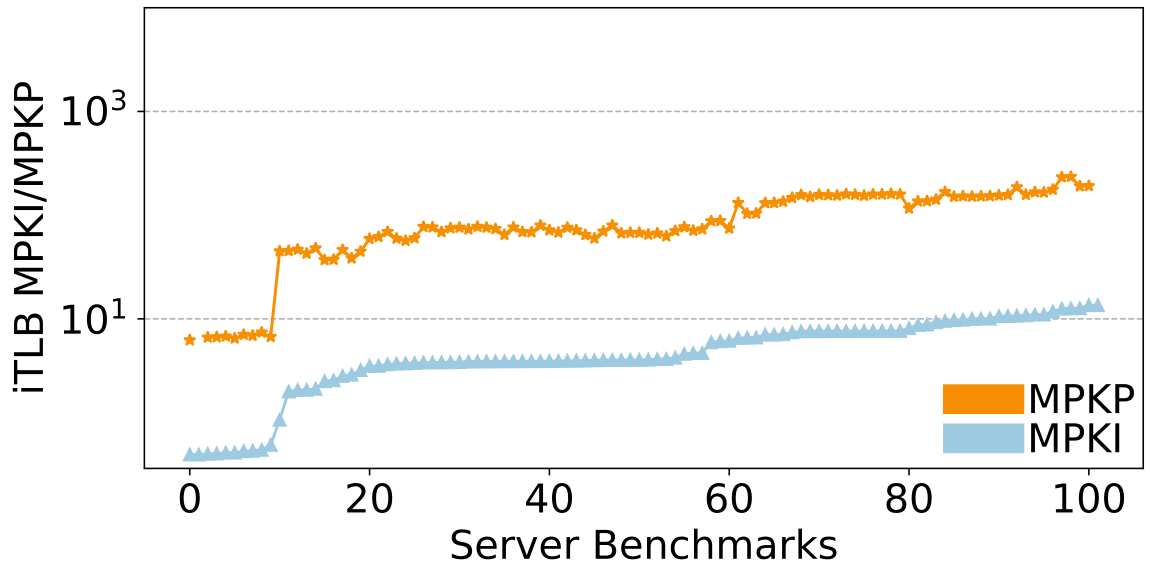


Figure 3.6: iTLB MPKI/MPKP

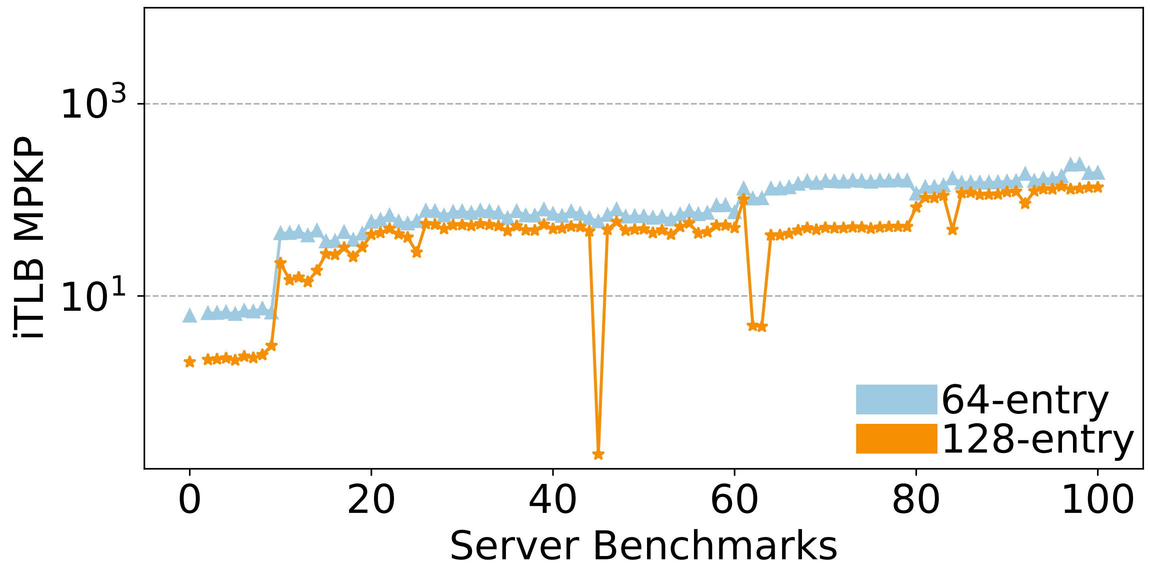


Figure 3.7: Varying iTLB Size

3.2.3 Frequent PTE Movement for the iTLB

A reasonably sized iTLB often fails to accommodate the active page table entries (PTEs) required by applications with larger code footprints, as we shall demonstrate. The canonical metric used to quantify the magnitude of iTLB misses is Misses Per Kilo Instructions (MPKI). We introduce an additional metric, *Misses Per Page Transition* (*MPKP*), which measures the misses occurring per kilo page transitions. A page transition counts the number of times the processor switches from processing instructions from one page to another. This metric is analogous to MPKA for blocks and helps account for processor implementations that buffer the contents of an instruction TLB entry until the processor transitions to fetching instructions from a different page.

We investigate the movement of instruction TLB entries from a lower-level TLB to a higher-level TLB (iTLB). This analysis is conducted for a 64-entry iTLB with a base page size of 4KB, as illustrated in Figure 3.6 (note the log scale on the y-axis). The iTLB MPKIs range from 0.5 to 13. Although the MPKIs are smaller compared to those of an L1i due to the larger granularity of operation, they remain substantial for a TLB. The MPKP is relatively high for many benchmarks, exceeding 100 (indicating that over 1 in 10 accesses involve the lower-level TLB) for these benchmarks, nearly an order of magnitude higher than the MPKIs. The large code footprint leads to frequent misses and the movement of instruction TLB entries.

Figure 3.7 examines the MPKP for both a 64-entry and a 128-entry iTLB. We observe that numerous applications continue to exhibit a high MPKP even with a 128-entry TLB size.

3.2.4 Impact of Large Code Footprint

All of this data collectively suggest that many of these applications frequently miss in the primary structures and involve a secondary structure during instruction fetch, even with relatively larger-sized primary structures.

A very large Branch Target Buffer (BTB) combined with an effective and large branch direction predictor significantly enhances the precise traversal of an application's control flow graph, thereby establishing an accurate instruction reference stream. Similarly, a large instruction cache or iTLB ensures the timely delivery of necessary instructions to the backend of the processor. However, the use of very large primary structures is generally impractical due to design complexity, energy consumption, and clock-cycle constraints.

This issue requires mechanisms to ensure that instructions and the associated information needed for processing are accessed from the memory hierarchy or secondary structures and moved to the primary structures before the processor requires them. The objective is to eliminate pipeline stalls caused by misses in primary structures. Key advancements in this domain can be broadly classified into three categories:

- **Instruction Prefetching (code blocks) schemes**, which involve techniques to preload blocks of instructions in the L1i based on predicted future accesses to minimize fetch latency and reduce associated bubbles in the pipeline. We discuss many such techniques in detail in Section 3.3.

- **BTB Management Techniques** which involve managing BTBs to ensure that they have the information necessary to correctly fetch instructions without stalling frequently. We discuss many such techniques in Section 3.4.

- **Instruction Translation Lookaside Buffer (iTLB) Management Schemes**, which encompass techniques for preloading address translations into the iTLB. These techniques aim to reduce instruction fetch translation latencies and reduce associated pipeline bubbles. A comprehensive analysis of these methodologies is provided in Section 3.5.

We describe each of these in detail next.

3.3 Instruction Prefetching of Code Blocks

Ensuring that blocks of instructions are loaded in the L1i in a timely manner is crucial for effective instruction fetching and minimizing pipeline bubbles. All instruction prefetchers aim to cover as many misses as possible, maximizing the miss coverage. In this process, they often trade off between two key metrics: **block supply accuracy** and **timeliness**, which we define next.

3.3.1 Key terms

- **Block supply accuracy** refers to the percentage of instruction code blocks supplied to the L1i that are referenced by the processor before the block is evicted from the cache.

- **Timeliness** refers to the scheme's capability to fully tolerate the latency associated with fetching instructions from a lower-level cache, minimizing the time spent waiting for an instruction.

3.3.2 Software Techniques

There have been various software prefetching techniques that typically involve profiling and instrumentation of programs. We briefly describe some of these techniques below, though we primarily focus on a detailed discussion of hardware-based (microarchitectural) solutions in this dissertation.

The use of helper threads, as seen in works like [77, 87], attempts to create compressed representations of a program and use idle cores or contexts to execute these representations. This method accelerates the main thread by precomputing branch outcomes and prefetching missing data blocks. The use of helper threads has also been explored for prefetching code blocks [4]. Such techniques require execution resources, such as extra hardware contexts, to execute these threads.

Other techniques [14, 37, 41, 49, 51, 83] involve the compiler to insert instruction prefetch instructions, which requires the compiler to have profile information. Additionally, these methods require ISA changes to include special instruction prefetch instructions. The execution of these inserted instructions takes up additional fetch and dispatch slots.

3.3.3 HW Prefetch Taxonomy Overview

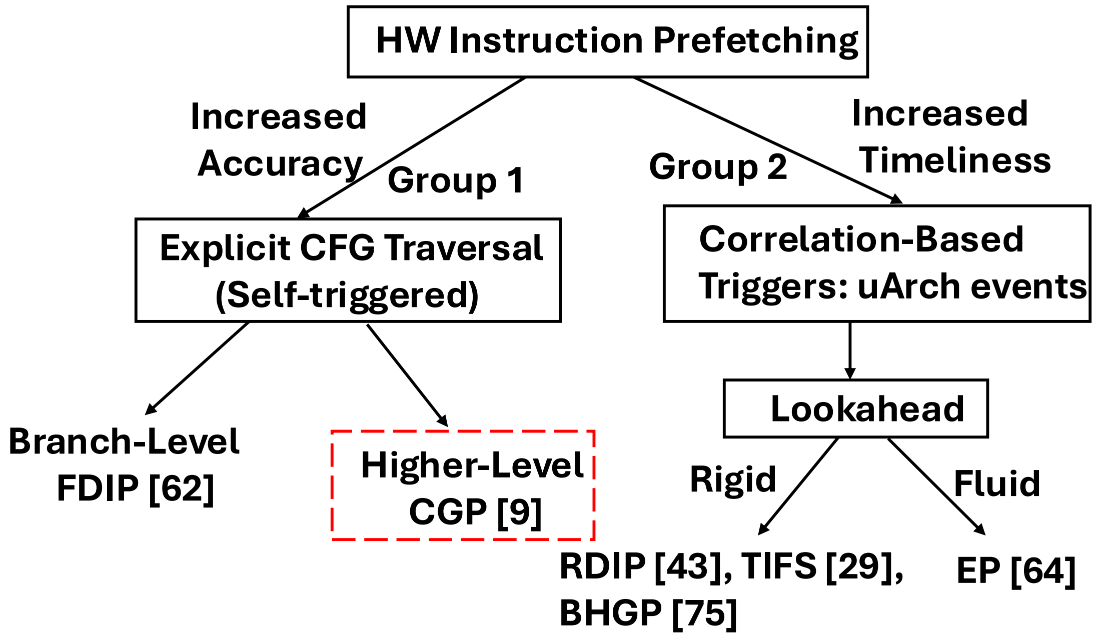

HW Instruction Prefetching
Increased Accuracy
Group 1
Group 2
Increased Timeliness
Explicit CFG Traversal (Self-triggered)
Correlation-Based Triggers: uArch events
Lookahead
Branch-Level FDIP [62]
Higher-Level CGP [9]
Rigid
Fluid
RDIP [43], TIFS [29], BHGP [75]
EP [64]

Figure 3.8: HW Prefetch Taxonomy

We first introduce a taxonomy, borrowed and expanded from [29, 64], to classify instruction prefetchers based on their operation, as shown in Figure 3.8.

The first group attempts to **explicitly traverse the program's control flow graph** to prefetch blocks along the predicted path. Some schemes in this group aim to accurately predict the outcome of every control instruction to determine the upcoming control flow. The key example is Fetch Directed Instruction Prefetching [62]. Other schemes in this group attempt to uncover upcoming control flow at a higher level, such as Callgraph Prefetching [9].

The second group attempts to **capture correlations between microarchitectural events and cache misses** and use these learned correlations to prefetch blocks that are likely to miss. A key parameter associated with such schemes is the lookahead parameter, which we describe in greater detail in Section 3.3.5. Some examples in this group include Return Directed Instruction Prefetching (RDIP) [43] and Entangling Prefetching [63].

We next describe each of these groups in greater detail.

3.3.4 Group 1

Fetch Directed Instruction Prefetching (FDIP)

First, we describe schemes that rely on accurately determining the outcome of every control instruction in the program, the main example being *Fetch Directed Instruction Prefetching* (*FDIP*) [35, 62] and its predecessors [24]. These prefetchers utilize the program counter (PC) determination logic, using the branch direction prediction, Branch Target Buffer (BTB), and Return Address Stack (RAS), to establish a precise instruction reference stream. This stream guides the fetching of required instruction blocks from memory. FDIP decouples these two components via a Fetch Target Queue (FTQ), allowing the PC determination logic (next instruction) to run ahead, thereby tolerating the latency of fetching instruction blocks from a lower-level cache. This approach typically requires large BTBs and accurate branch predictors to establish an effective instruction reference stream for effective prefetching.

Redirects refer to events where the instruction fetching logic (PC determination logic) goes down an incorrect path due to branch direction misprediction, BTB miss, or a RAS mispredict. More generally, for a scheme sequencing program control flow, a redirect occurs when the scheme incorrectly sequences the program control flow, resulting in the re-steering of the sequencing process along the correct path. In the case of FDIP, redirects lead to flushing the FTQ and restarting the PC determination logic from the target of the mispredicted or missed control instruction.

Sequencing Accuracy - For a scheme traversing program control flow, sequencing accuracy is the ability to sequence the program control flow without being redirected. The lower the number of redirects, the higher the sequencing accuracy. High sequencing accuracy implies that the scheme can stay on track most of the time.

Block Supply Accuracy for FDIP is closely tied to its sequencing accuracy, which has significantly improved through the adoption of designs featuring large BTBs and highly accurate branch direction predictors. These advancements collectively enable the precise discovery of instruction streams and high block supply accuracy, as demonstrated in recent studies [35].

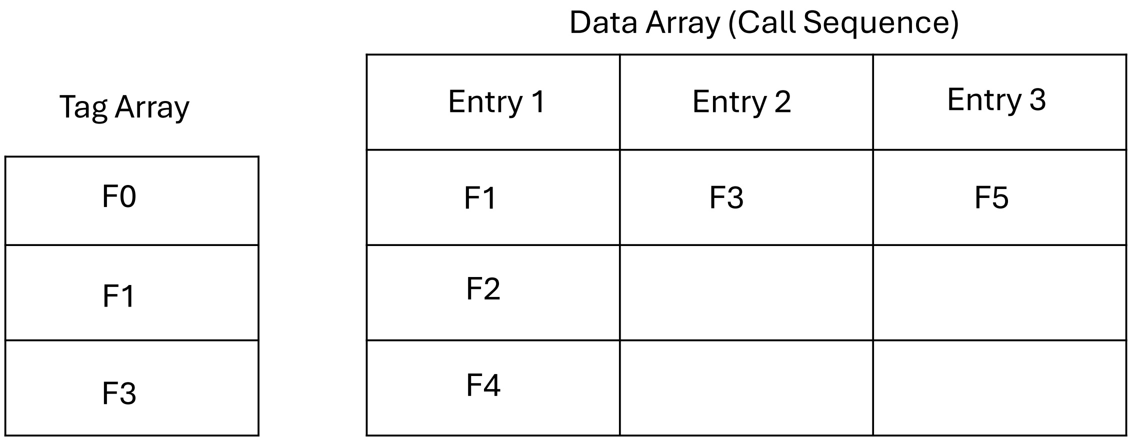
Timeliness for FDIP is tied to its ability to maintain sufficient decoupling from the instruction fetching process, allowing it to tolerate the latency of fetching instruction blocks from a lower-level cache. Redirects cause FDIP to slow down because they result in flushing the FTQ and restarting the PC determination logic from the target of the mispredicted or missed control instruction. In the absence of redirects, FDIP with sufficient decoupling can stay sufficiently ahead and ensure a more timely delivery of instructions.

Callgraph Prefetching (CGP)

```
F0(){     F1(){     F2(){     F3(){     F4(){     F5(){
.....     .....     .....     .....     .....     .....
F1();     .....     .....     .....     .....     .....
....      F2();     return;   F4();     return;   return;
...       ....      }         ....      }         }
F3();     return;             return;
....      }                   }
F5()
....
return
}
```

Figure 3.9: Program for Callgraph Construction

Next, we describe a scheme that traverses high-level control flow in a limited manner. *Callgraph Prefetching (CGP)* [9] is a technique that seeks to traverse program control flow at the call graph level. The key insight behind this proposal is



	Data Array (Call Sequence)		
Tag Array	Entry 1	Entry 2	Entry 3
F0	F1	F3	F5
F1	F2		
F3	F4		

Figure 3.10: Call Graph History Table

that most functions have relatively stable call sequences. For example, if function A calls functions B and C, this sequence of function calls is likely to remain stable across multiple invocations of function A. CGP exploits this insight by capturing these call sequences for the different functions in the program and using this program representation.

Consider the example shown in Figure 3.9, where we have six functions: F0 through F5. Function F0 calls F1, F2, and F3. Function F1 calls F2. Function F3 calls F4. Functions F2, F4, and F5 do not make any function calls. During the program's execution, the processor learns the Call Graph History Table (CGHT) as illustrated in Figure 3.10.

Figure 3.10 shows the learned callsequences for the different functions. The call sequence for function F0 includes the addresses of functions F1, F3, and F5. The call sequence for function F1 contains only the address of function F2. The call sequence for function F3 includes the address of function F4. Functions F2, F4, and F5 do not have entries as they do not make any function calls.

When the processor starts fetching instructions for a function, the idea is to use the learned call sequence to prefetch code blocks for the next function to be processed. For functions that call multiple other functions (such as F0 in Figure 3.10), CGP uses the processor's Return Address Stack (RAS) to maintain state, ensuring it knows which function to prefetch when the processor returns to the calling function.

Block Supply Accuracy for CGP is closely tied to the accuracy in correctly predicting the next function. It slightly underperforms compared to a scheme like FDIP because it traverses high-level control flow without accurately predicting the outcome of every control instruction, resulting in the movement of extra blocks. CGP prefetches only a fixed number of contiguous blocks following the function to prefetch, in order to limit the excess supply of blocks.

Timeliness of CGP is superior to FDIP because it can bypass multiple control instructions when sequencing high-level control flow. However, CGP's timeliness is constrained by its ability to prefetch only parts of the next function to be called and its tight coupling to the structures used for fetching instructions, particularly the Return Address Stack (RAS). Although this aspect has potential for enhancement, it is not explored by the authors in this work.

Both FDIP and CGP are tied to the logic involved in fetching instructions. We next describe this coupling in more detail.

Coupling to Instruction Fetch Logic

Redirects within the instruction fetching logic are crucial to FDIP's performance. Although CGP creates a program representation independent of the logic used for fetching instructions, its operation is closely tied to this logic. Consequently, frequent redirects within the instruction fetching logic hinder the performance of both FDIP and CGP.

Sources of redirects within instruction fetching logic include:

- Branch direction mispredicts
- BTB misses
- RAS mispredicts

The magnitude of RAS mispredicts is much smaller, so we focus on the first two sources. We use the metric *Redirects Per Kilo Instructions* (*Redirects per KI*) to quan-

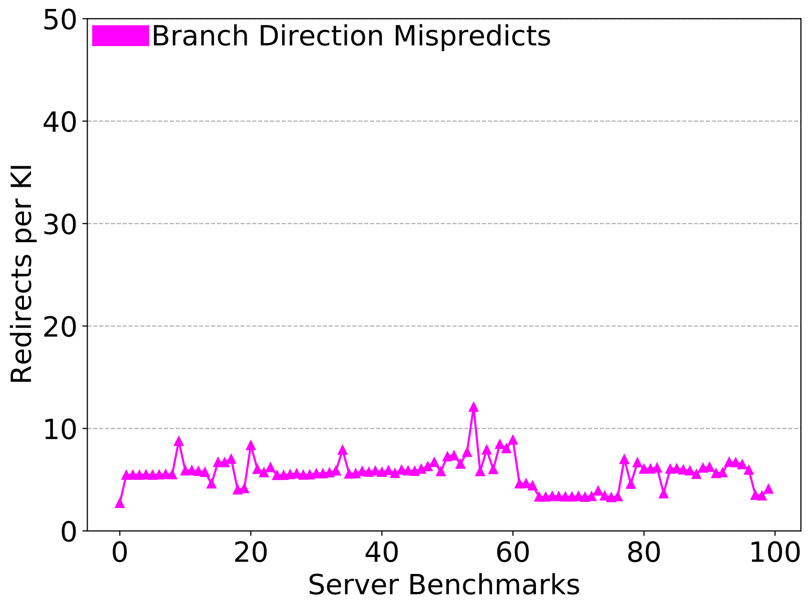


Figure 3.11: Redirects from Branch Direction Prediction

tify the frequency of redirects. Figure 3.11 quantifies the Redirects per KI due to branch direction mispredicts alone, using a well-performing direction predictor. Direction predictors predict the path taken by a conditional branch, and a misprediction occurs when the prediction does not match the actual outcome, which is known after the branch has executed. The redirects are mostly in the lower single digits, and for some benchmarks, slightly over ten. Figure 3.12 quantifies the total Redirects per KI for different BTB sizes. We observe that Redirects per KI with a 512-entry BTB are over 20 for many benchmarks. Increasing the BTB size to 2K entries significantly reduces the redirects for some benchmarks, and increasing the size to 8K entries reduces them further, bringing the redirects down by an order of magnitude, close to the level of an infinite BTB in most cases. The redirects with an infinite BTB are the same as what we observed in Figure 3.11, primarily resulting from direction mispredicts. This suggests that completely relying on the next PC determination logic to uncover upcoming control flow is likely to require large BTBs (or separate BTB management) to cope with the growing application code

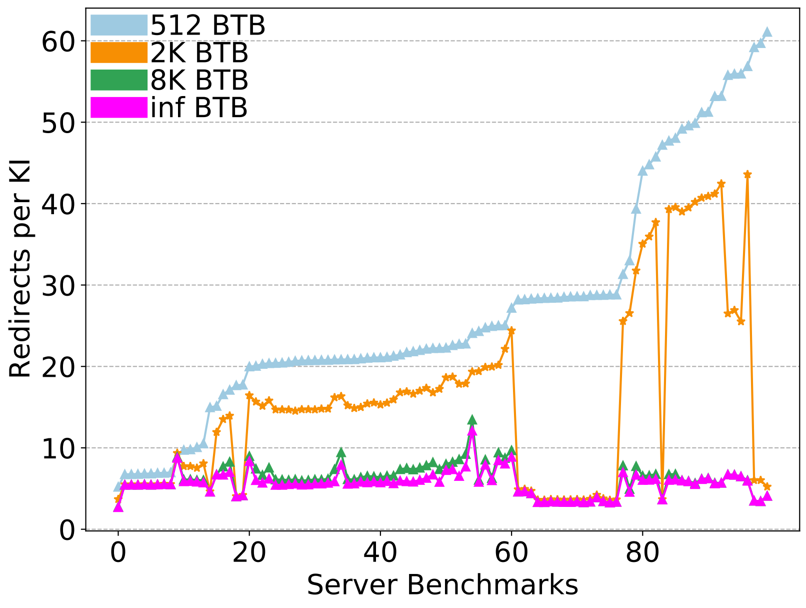


Figure 3.12: Redirects (Varying BTB Size)

footprint.

It is therefore not surprising that modern implementations have moved to using significantly larger BTBs. The Zen5 microarchitecture [3], for example, has a large 16K-entry BTB. Realizing such large BTBs is challenging, as processors rely on making one or more branch predictions every cycle to facilitate the creation of large instruction windows. Processor designers typically address this challenge by pipelining the BTB.

Many implementations also employ a smaller BTB (which is accessed in one cycle) corrected by a larger BTB that takes multiple cycles to access (2-3 cycles) [5, 66]. Both these BTBs are accessed in parallel. Having a large BTB that is accessed every cycle (or every other cycle) is likely to result in increased energy consumption, in addition to the design complexity it brings. One example of such a design is the Samsung M3 [66], which employs a 128-entry micro BTB corrected by a 4K-entry primary BTB. It also uses a 16K-entry secondary BTB. Another example is the Arm Neoverse N2 [60], which employs a 64-entry nano BTB corrected by an

8K-entry primary BTB.

An infinite BTB, a perfect direction predictor, and infinite bandwidth PC determination logic would enable FDIP, with adequate decoupling, to ensure timely delivery of blocks to the L1i and instructions to the backend of the machine. However, real implementations are more constrained, and these constraints result in increased redirects within the processor frontend. This limits FDIP's ability to advance further in the instruction stream, thereby limiting the timely delivery of blocks to the L1i. The requirement to accurately go past every control instruction in the program is challenging in the presence of direction mispredicts and BTB misses.

CGP captures a high-level program representation, eliminating the need to accurately determine the outcome of every control instruction in the program. However, the operational design of CGP limits its effectiveness by tying it to the limitations of the instruction fetching logic. Fundamentally, CGP's operation does not need to be tied to the logic used for fetching instructions. It could use a separate return address stack to traverse the call graph, remaining decoupled from the instruction fetching logic. Additionally, CGP could be enhanced to supply discontiguous blocks within functions, thereby improving the supply of blocks to the processor.

3.3.5 Group 2

Next, we describe the second group of prefetchers that seek to capture correlations between microarchitectural events and cache misses and use this information to facilitate prefetches. The learned correlations are used to trigger the prefetching of blocks.

Block Supply Accuracy is the metric that these schemes often compromise compared to Group 1 prefetchers. These prefetchers try to more aggressively get ahead in the instruction stream using correlations and eagerly move cache blocks to maintain timeliness.

Timeliness is the key metric optimized by these schemes to outperform Group

1 prefetchers. The motivation is to go past multiple control instructions in one step and get ahead in the instruction stream using established correlations with earlier microarchitectural events.

Lookahead refers to the distance a scheme keeps ahead of the processor. This distance can be measured in various units, such as cache blocks, basic blocks, or other parameters specific to the scheme. It is a key metric for these prefetchers, comparable to sequencing accuracy in Group 1 prefetchers, as the schemes in this group strive to maintain higher timeliness

For example, a scheme might use basic blocks as triggers to initiate the prefetching of blocks. If it links basic blocks to cache blocks accessed five dynamic basic blocks later, then the scheme effectively maintains a lookahead of five basic blocks. Prominent examples include [4, 10, 30, 34, 43, 59, 64, 69, 74, 75, 86], which use different events to capture correlations. Most schemes in this group employ a fixed (rigid) lookahead, which we describe first. However, the state-of-the-art scheme in this group [64] employs a more fluid lookahead to ensure even more timely prefetching of blocks, which we describe later. This is not to say that other schemes cannot have a fluid lookahead; rather, it was a key contribution of [64].

Rigid Lookahead Schemes

One-block lookahead [69] starts to prefetch block B+1 when block B experiences a miss, with the miss to block B being the event triggering the prefetch of block B+1. Selective Next-4 line prefetching [10] continues to be triggered by cache misses and selectively prefetches blocks B+1, B+2, B+3, and B+4 when block B experiences a cache miss.

Branch History Guided Prefetching (*BHGP*) [75] is a scheme where the key idea is to establish a correlation between the missed basic block and the execution of an earlier conditional branch, keeping a static number of branches ahead. Establishing such correlations also allows the scheme to go past multiple branches to prefetch. When executing a branch, the table is looked up to trigger a timely prefetch of cache blocks corresponding to the appropriate basic block.

Temporal Instruction Fetch Streaming (*TIFS*) [29] is another proposal to prefetch temporally correlated instruction miss streams from a lower-level cache. Most misses are part of recurring streams. The key idea is to record and replay repeating streams of misses. TIFS records such streams and uses access to a block in the stream to trigger a prefetch of the subsequent block, maintaining a fixed number of misses ahead in the stream. Any deviation results in prefetching from a new stream. TIFS establishes correlations among missing blocks and uses these correlations to drive the prefetching of blocks.

Proactive Instruction Fetch (*PIF*) [30] is an extension of TIFS designed to reduce the storage requirements of TIFS by using bit vectors to encode blocks in a region. Additionally, PIF employs temporal compaction to avoid storing the same addresses repeatedly in the presence of constructs such as loops. PIF records the committed instruction sequence and can tolerate disruptions from wrong-path execution.

Return Address Stack Directed Instruction Prefetching (*RDIP*) [43] is a more recent prefetching technique that associates prefetch operations with signatures derived from the contents of a Return Address Stack (RAS). We describe this technique in detail next.

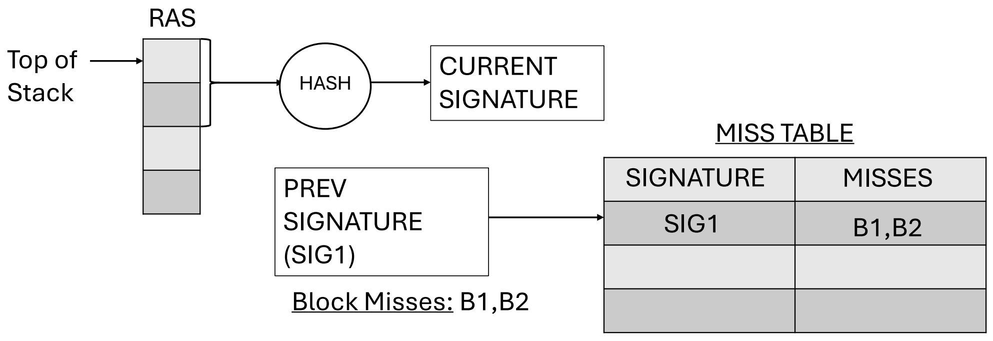


Figure 3.13: RDIP Datastructures

As shown in Figure 3.13, RDIP generates a hash of the first n (typically 2-4) entries of the return address stack (RAS), referred to as a signature. It establishes a

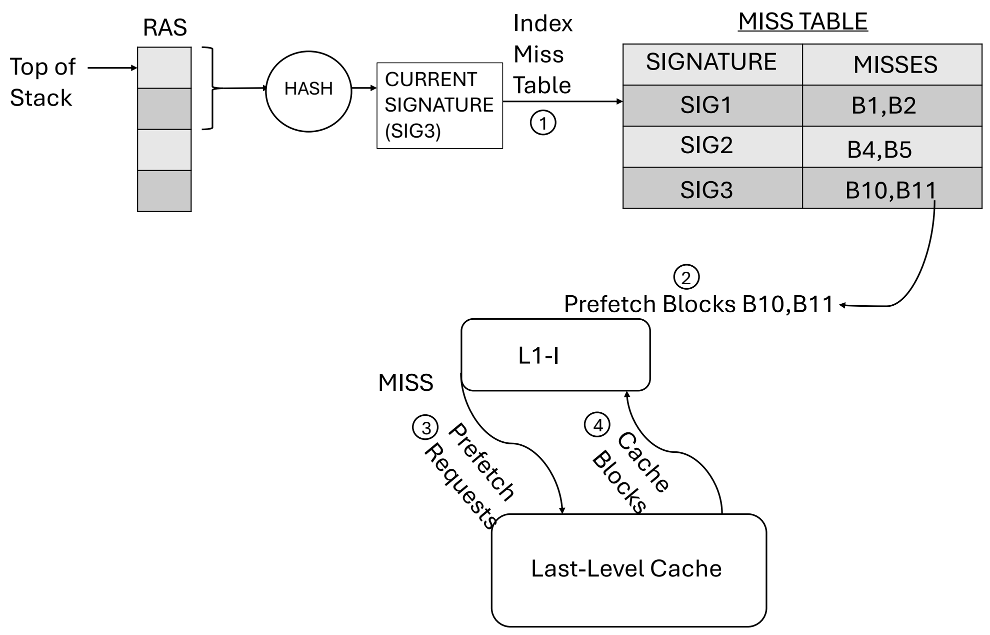


Figure 3.14: RDIP Prefetching

correlation between the preceding signature and any cache blocks that miss in the L1i. For example in Figure 3.13, it establishes a link between Sig1 and the missing cache block addresses B1 and B2. The correlation is made with the preceding signature, which serves as the prefetch trigger. RDIP uses these established correlations to prefetch instruction cache blocks, indexing the table using the RAS-based signature to trigger prefetches, as shown in Figure 3.14. In this example, Sig3 is used as an index in the table to prefetch blocks B10 and B11 into the L1i.

RDIP always maintains a lookahead of one signature, which can tolerate the miss latency when very few instructions are executed as part of a signature. Additionally, using multiple entries to produce a hash increases the number of signatures when functions have multiple callers, often duplicating the same miss information in multiple entries.

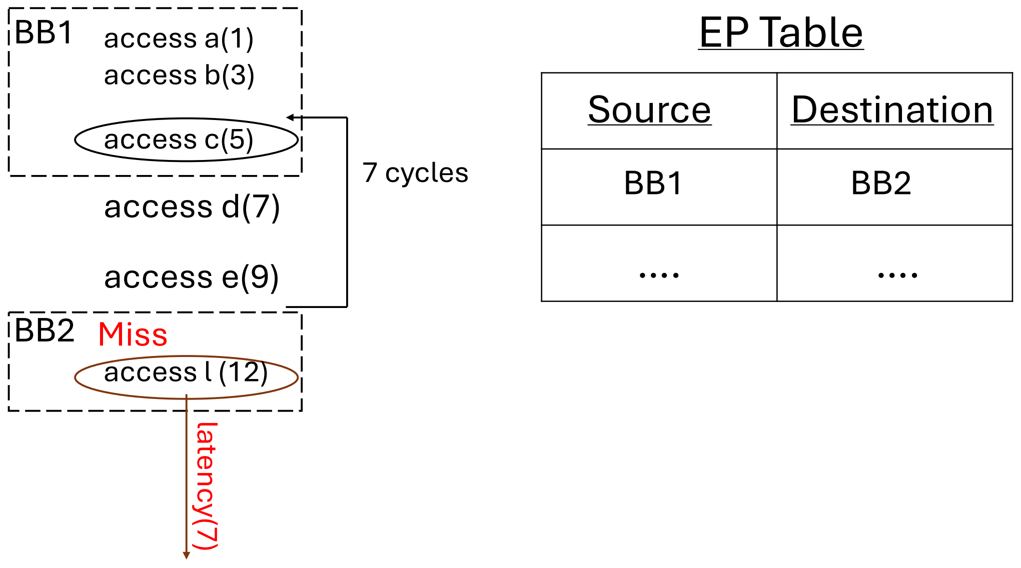


Figure 3.15: EP Datastructures

Fluid Lookahead Schemes

The highest performing prefetcher in this category is the *Entangling Prefetcher* (*EP*) [63, 64], which was proposed recently. It correlates cache misses to prior cache access events. Unlike other schemes described earlier, EP establishes correlations in a timeliness-aware fashion, rather than employing a fixed or a static lookahead.

EP establishes correlations between basic blocks (BB), specifically between a basic block that experiences a miss in the instruction cache and a basic block that initiated an instruction cache access earlier. As shown in Figure 3.15, EP establishes a correlation between a prior cache access made to basic block 1 (BB1) and the cache misses occurring in basic block 2 (BB2). This approach is designed to tolerate the latency of accessing BB2 from a lower-level cache, and it stores this timeliness-aware correlation in an EP table. EP subsequently uses this learned information from the EP table, as shown in Figure 3.16, to initiate a timely prefetch when cache blocks that are part of BB1 are accessed.

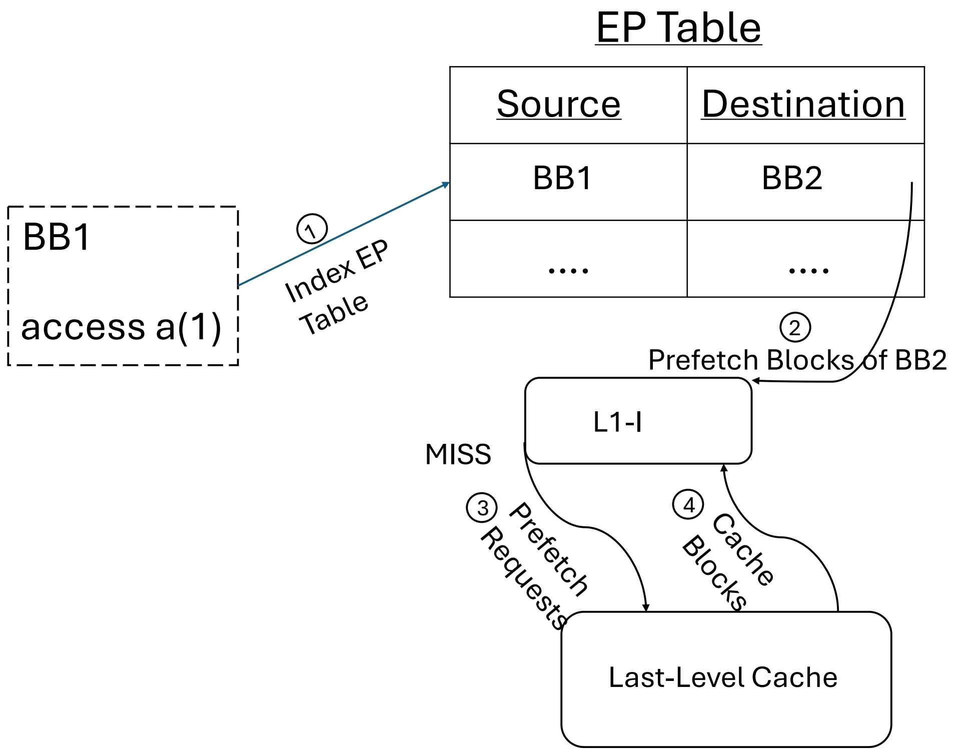


Figure 3.16: EP Prefetching

Coupling to Instruction Fetch Logic

BHGP [75] explicitly relies on the branch predictor (part of the PC determination logic) to establish correlations. RDIP [43] relies on the RAS (part of the PC determination logic) to establish correlations. Other schemes rely on L1i accesses or L1i misses to establish correlations [29, 30, 64]. While these schemes are not explicitly tied to parts of the PC determination logic, they are implicitly tied to it because the PC determination logic creates the instruction reference stream, resulting in L1i accesses and misses. As a result, most schemes in this group remain closely coupled with the instruction fetch process.

3.4 Instruction BTB Management

Having the requisite information within primary branch prediction structures to correctly fetch instructions is essential for maintaining an effective supply of instructions to the backend of the processor. Large code footprint applications, characterized by frequent code movement between secondary and primary caches, present a challenge, as retaining all necessary information becomes difficult with reasonably sized primary structures. We present a taxonomy of BTB management schemes designed to address this issue.

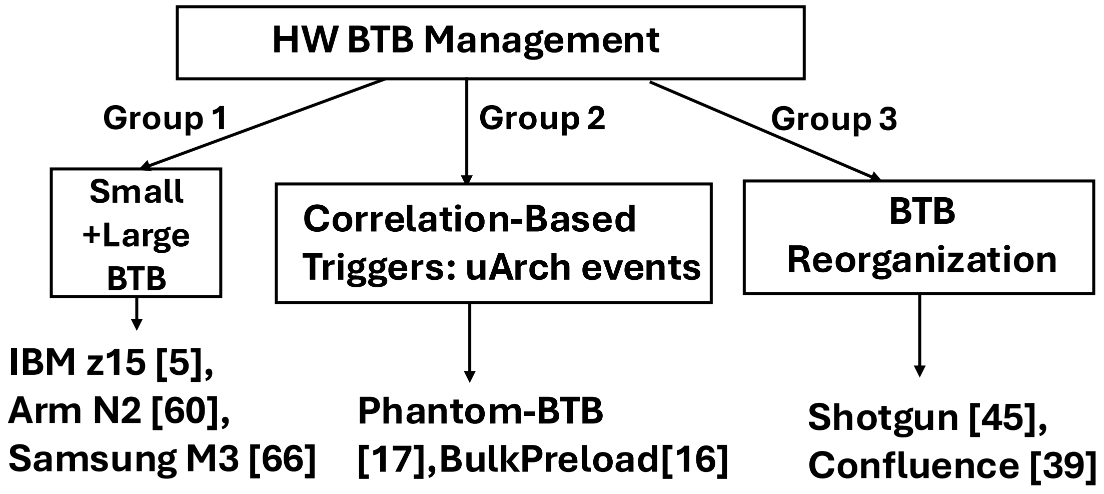

HW BTB Management
Group 1
Group 2
Group 3
Small +Large BTB
Correlation-Based Triggers: uArch events
BTB Reorganization
IBM z15 [5], Arm N2 [60], Samsung M3 [66]
Phantom-BTB [17],BulkPreload[16]
Shotgun [45], Confluence [39]

Figure 3.17: HW BTB Management Schemes

3.4.1 HW BTB Management Taxonomy Overview

Figure 3.17 illustrates a taxonomy of schemes designed to manage a Branch Target Buffer (BTB). The first group consists of designs that employ both a small and a large BTB, where the smaller BTB is corrected by a larger BTB. Examples of this include commercial processor designs such as the IBM z15, ARM N2, and Samsung M3 [5, 60, 66]. The second group consists of correlation-based BTB prefetching schemes that correlate microarchitectural events with BTB misses and use those events to trigger the prefetch of BTB entries into an L1 BTB. Examples include [16, 17, 18]. The third group consists of schemes that attempt to reorganize BTBs to reduce BTB misses, with examples such as Shotgun and Confluence [39, 45].

We next describe each of these groups in greater detail.

3.4.2 Group 1

The first group, commonly employed in modern processors, involves the use of a small BTB corrected by a larger BTB [5, 60, 66], with both being accessed in parallel. The smaller BTB is typically accessed within a single cycle, whereas the larger BTB usually requires 2-3 cycles. Accessing a larger BTB entails increased design complexity, necessitating the pipelining of the large BTB and resulting in higher energy consumption. However, this approach offers the advantage of retaining target addresses for a large number of branches, thereby reducing fetch redirects. This technique has gained popularity among processor implementations due to its effectiveness.

3.4.3 Group 2

The second group of schemes correlate BTB misses with other microarchitectural events and uses those events to trigger the prefetching of BTB entries into the BTB. For example, predictor virtualization [16, 17, 18] aims to provide the appearance of large-sized processor front-end structures while operating with small-sized physical structures. Specifically, BTB virtualization correlates a group of BTB misses with a prefetch trigger (such as a BTB access) and moves this group to the small-sized primary structure (L1 BTB) when triggered by a BTB access.

3.4.4 Group 3

The third group of schemes attempts to reorganize the BTB and prefill it in a just-in-time manner. The main examples in this group are Shotgun [45] and Confluence

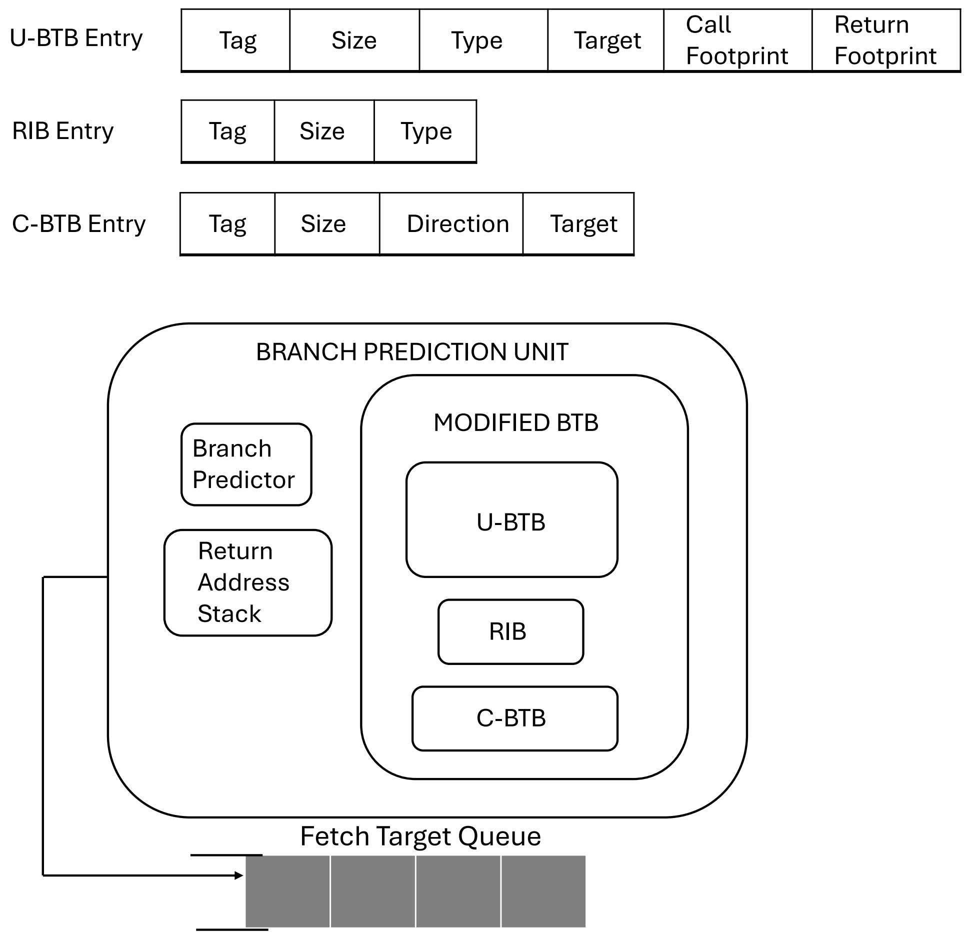


Figure 3.18: Shotgun Datastructures

[39]. Shotgun modifies FDIP by reorganizing the existing BTB into multiple components, making it more effective than a standard BTB of the same size, as shown in Figure 3.18.

An Unconditional BTB (UBTB), based on high-level control flow, is designed to retain all target addresses for the active unconditional control instructions (calls and jumps) in the program. As shown in Figure 3.18, a UBTB entry stores a tag to identify the branch, a size field that records the size of the basic block containing the branch, and a type field that specifies the branch type. For calls and uncon-

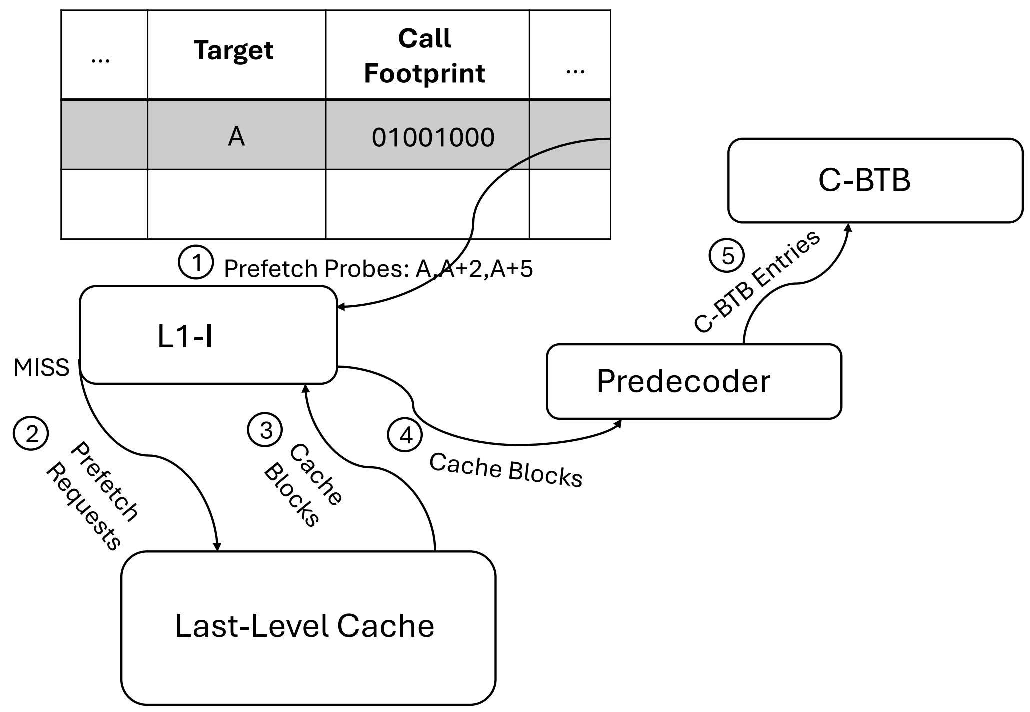


Figure 3.19: Shotgun Prefetching

ditional jumps, the UBTB maintains a spatial footprint of blocks, encoded as the first block address and a bit vector indicating the cache blocks accessed following the instruction until the next call, unconditional jump, or a return. For call instructions, it also maintains a return block footprint associated with the return to the instruction following the call site.

A Return Instruction Buffer (RIB) is used to identify return instructions, with the target address obtained from the return address stack. The block footprint in the UBTB is used to prefetch into the L1i and fill a primary conditional BTB (CBTB) just-in-time, as depicted in Figure 3.19. Shotgun does not rely on the CBTB to generate the instruction reference stream for prefetching; instead, it uses the block footprint for this purpose, reserving CBTB entries solely for correctly resteering the flow of instructions after the relevant control instructions have been fetched into the L1i. An entry in the CBTB simply stores the identity of the conditional branch and the associated target and last predicted direction.

The idea is to downsize the CBTB because it is filled in a just-in-time manner, allowing Shotgun to operate effectively with relatively smaller-sized BTBs. However, Shotgun still requires a large UBTB to retain the target addresses of all unconditional branches and continues to depend on the PC determination logic to go past every control instruction in the program.

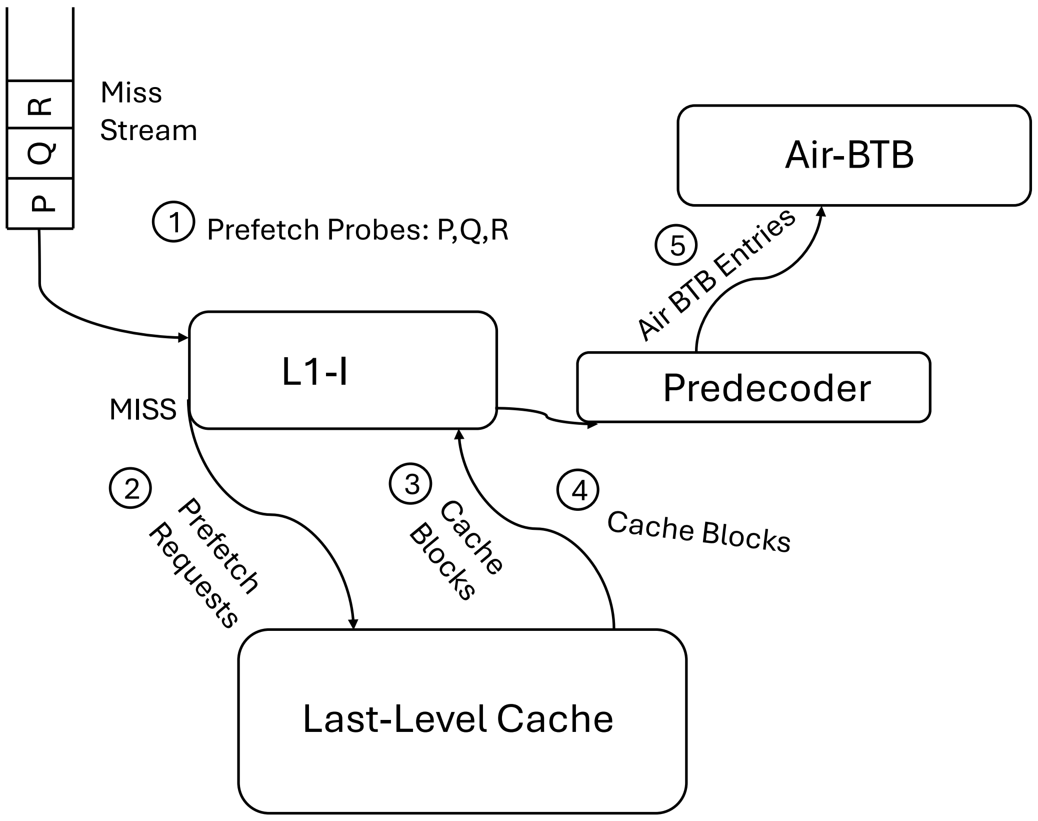


Figure 3.20: Confluence Operation

Confluence [39] reorganizes the BTB and leverages temporal cache block miss streams, similar to TIFS [29], as explained earlier and shown in Figure 3.20. The key idea behind TIFS is to record and replay repeating streams of cache block misses. In the Figure 3.20, we see that the recorded miss stream containing miss addresses P, Q, and R is used for prefetching. Confluence extends this to pre-fill a reorganized BTB (AirBTB) in addition to the L1i. It employs predecode logic to generate BTB entries as code cache blocks are moved to the L1i, installing the entries in the Air-BTB.

3.5 Instruction TLB Management Schemes

The small size of L1 TLBs (iTLBs), typically 32 or 64 entries, makes prefetching into these structures less common due to the risk of polluting these structures. Recent research, such as [80], focuses on prefetching instruction TLB entries into the L2 TLB for applications that frequently encounter instruction TLB misses in the secondary TLB, which are primarily caused by interference with data translations.

However, as high-performance CPU designs have transitioned towards larger L2 TLB backing stores that are separated for instructions and as noted in [28], the interference with data TLB entries in the L2 TLB is significantly reduced. For most applications, a reasonably sized split L2 TLB for instructions can accommodate the majority of active instruction translations, rendering the prefetching of instruction TLB entries into the L2 TLB less critical. Despite this, there remains a crucial need to efficiently move translations into the iTLB to facilitate effective instruction fetching.

3.6 Potential for a Unified Scheme

The need for requisite information in various primary structures for effective instruction supply has led to different techniques that manage primary structures such as iTLBs, BTBs, and L1is, often separately. This results in designs that are an agglomeration of various techniques.

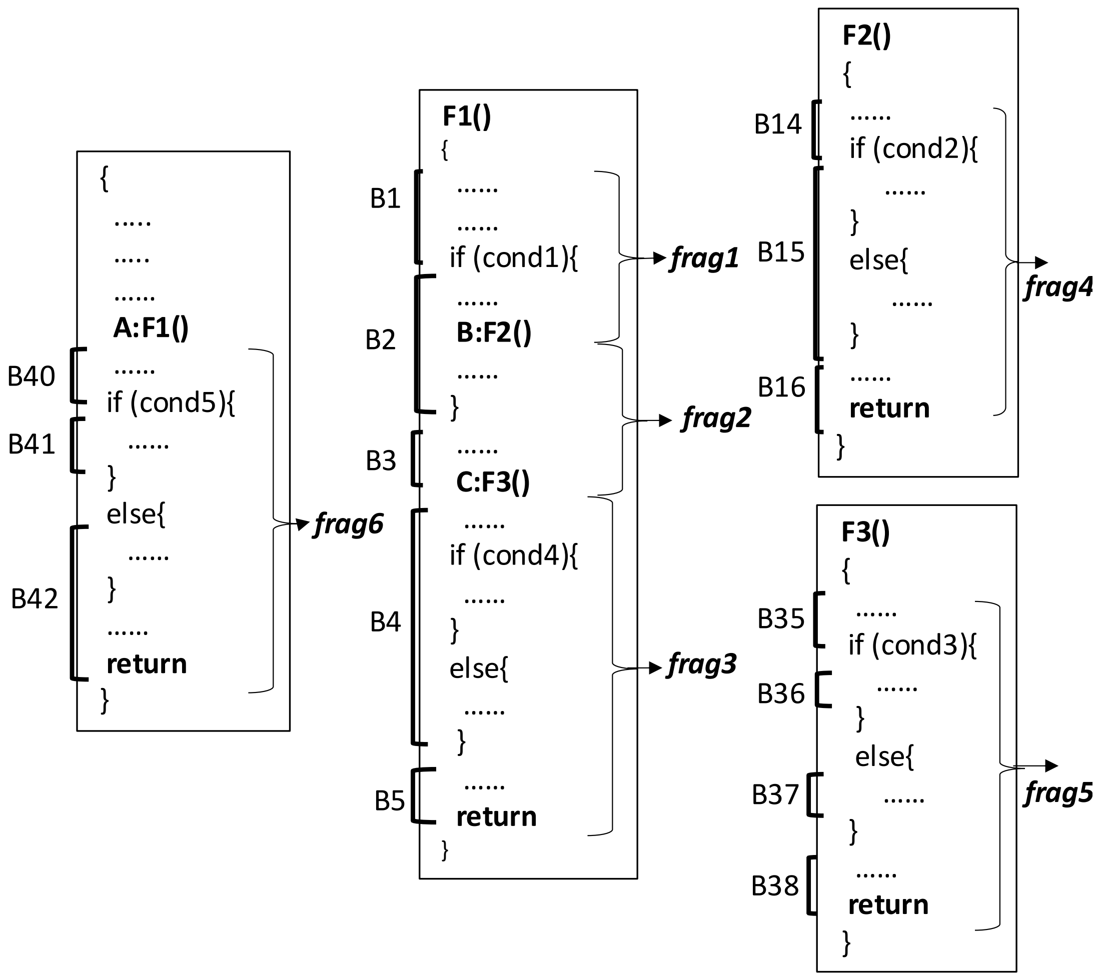
Fundamentally, for a scheme that does not rely upon or is not coupled to the PC determination logic to prefetch, both a BTB entry and an iTLB entry are relevant for correctly fetching instructions present in the L1i.

The problem we aim to solve is keeping the L1i supplied with the blocks the processor is likely to need, while also ensuring other primary structures (BTB and iTLB) are filled with the requisite information necessary to process instructions in these blocks. We are not focused on the precise instruction stream (next PCs) but rather a near precise block reference stream. BTB and iTLB entries are related to the static instructions in the block, so knowledge of a block reference stream can be used to fill reasonably sized BTBs and iTLBs just in time for processing instructions in the blocks.

One wonders if it is possible to develop a highly decoupled scheme—decoupled from the PC determination logic or the logic for fetching instructions—that can simultaneously keep the L1i supplied with the necessary blocks of instructions and ensure primary structures such as the BTB and iTLB are filled with the requisite information needed to process the instructions in these blocks that the processor is likely to reference. Can we leverage high-level control flow information to facilitate this movement, given that high-level control flow is likely unchanging and easier to capture and work with for this purpose? This approach could potentially help break the dependence on the logic for fetching instructions to uncover the blocks likely to be referenced by the processor. Furthermore, it could aid in filling reasonably sized primary structures, such as the BTB and iTLB, with the requisite information needed to generate the precise instruction stream in a just-in-time manner. We present an empirical rationale next to show why this might be possible and worth considering.

3.7 High-level Sequencing Potential (Fitting a Scheme in Group 1- Explicit CFG Traversal)

Fragment or a ***static fragment*** refers to a part of the program that starts at the target of a call and ends at another call or a return, representing a portion of the dynamic instruction stream delineated by calls and returns. A program can be partitioned into static fragments. A static fragment contains multiple branches and control instructions. It constitutes the blocks of instructions that are executed for all possible branch outcomes within the static fragment during the program's



```
{
 .....
 .....
 ......
 A:F1()
 ......
 if (cond5){
   ......
 }
 else{
   ......
 }
 ......
 return
}
```

```
F1()
{
  ......
  ......
 if (cond1){
  ......
  B:F2()
  ......
 }
  ......
  C:F3()
  ......
 if (cond4){
  ......
 }
 else{
  ......
 }
  ......
  return
}
```

```
F2()
{
 ......
 if (cond2){
   ......
 }
 else{
   ......
 }
 ......
 return
}
```

```
F3()
{
 ......
 if (cond3){
   ......
 }
 else{
   ......
 }
 ......
 return
}
```

Figure 3.21: Example Program Snippet

execution.

The execution of a program results in multiple dynamic instances of these static fragments, referred to as ***dynamic fragments***. These dynamic fragments may access some or all of the instruction blocks that are a part of the static fragment, depending on the outcomes of the branches within the fragment. Notably, different dynamic instances of the same static fragment can access different blocks of the static fragment.

More importantly, control flow at the fragment level is agnostic to the precise outcomes of the branches and other control instructions within the fragment. While determining the exact outcome of every control instruction is crucial for identifying the precise instruction stream to execute, it is less critical for determining which blocks the processor is likely to reference. Control flow at the fragment level is primarily concerned with determining the next fragment the processor is likely to execute.

Fragments starting at the target of a call are referred to as ***call fragments*** and fragments starting at the target of a return are referred to as ***return fragments***. Figure 3.21 provides an example to illustrate the concepts introduced above using a program snippet. The program snippet also highlights the code cache blocks to which the lines of the static program belong.

In this example, Function F1 is called from call site A to execute fragment frag1, composed of code cache blocks B1, B2, and B3. Function F2 is called at site B, depending on the evaluation of the if-condition, to execute fragment frag4, composed of code cache blocks B14, B15, and B16. Function F3 is called at site C to execute frag5, composed of code cache blocks B35, B36, B37, and B38. If the first fragment is frag1, the next fragment processed is either frag4 or frag5. Following the return from Function F2, the next fragment processed is the return fragment of call site B, which is frag2, composed of code cache blocks B2 and B3. Note that B2 and B3 are part of both frag1 and frag2. Following the return from Function F3, the next fragment processed is the return fragment of call site C, which is frag3, composed of code cache blocks B4 and B5. Following the return from Function F1, the next fragment processed is the return fragment of call site A, which is frag6, composed of code cache blocks B40, B41, and B42. Furthermore, we see that multiple fragments enclose if-then-else constructs. Frag1 through frag6 represent the static fragments for this program.

One dynamic instance of frag5 contains the blocks of instructions referenced when condition cond2 evaluates to true (B35, B36, and B38), while another dynamic instance of frag4 might contain the blocks of instructions referenced when cond2 evaluates to false (B35, B37, and B38). Regardless of the condition's outcome, we observe that the successor fragment for different dynamic instances of frag5 is always frag3.

However, in the case of frag1, one dynamic instance might result in cond1 evaluating to true, leading to the execution of blocks of instructions B1 and B2 and resulting in the successor fragment frag4. Another dynamic instance could involve cond1 evaluating to false, causing frag1 to access blocks along the not-taken path (B1 and B3) and resulting in a different successor fragment, namely frag5. This illustrates an example of how local control flow can result in different blocks of instructions being referenced and different fragment-level control flow.

3.7.1 Static Fragments

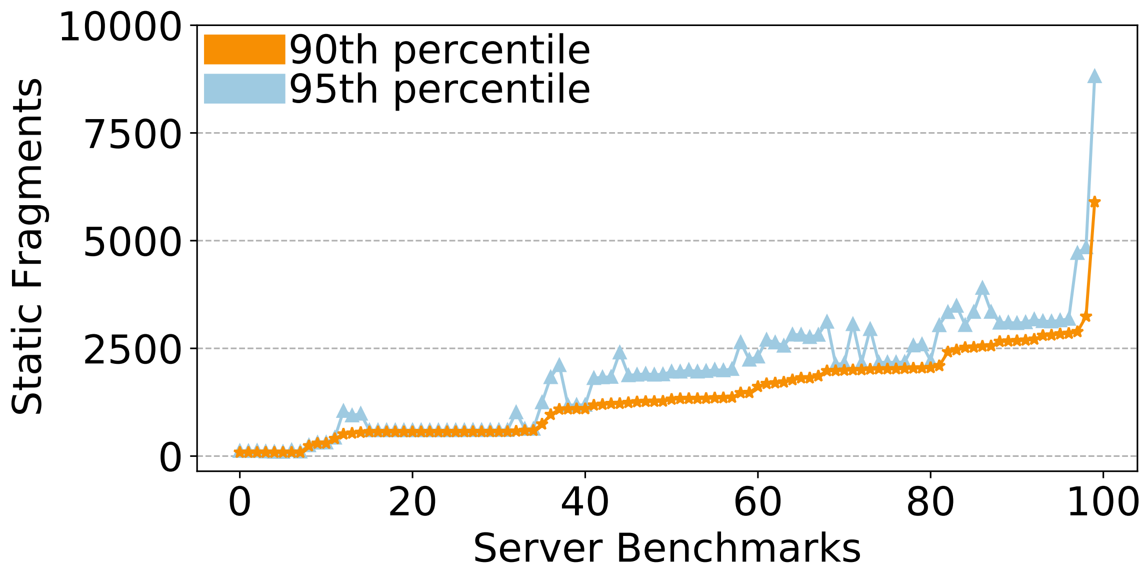


Figure 3.22: Static Fragments

Figure 3.22 illustrates the number of static fragments that account for 90% and 95% of program execution. Nearly all benchmarks have fewer than 2,000 fragments that account for 90% of program execution, with only a few exceeding this threshold. The number of fragments accounting for 95% of execution follows a similar trend, with the count being slightly higher than that for 90%. In Table 3.1, we present the fragments that account for 95% of program execution for each benchmark.

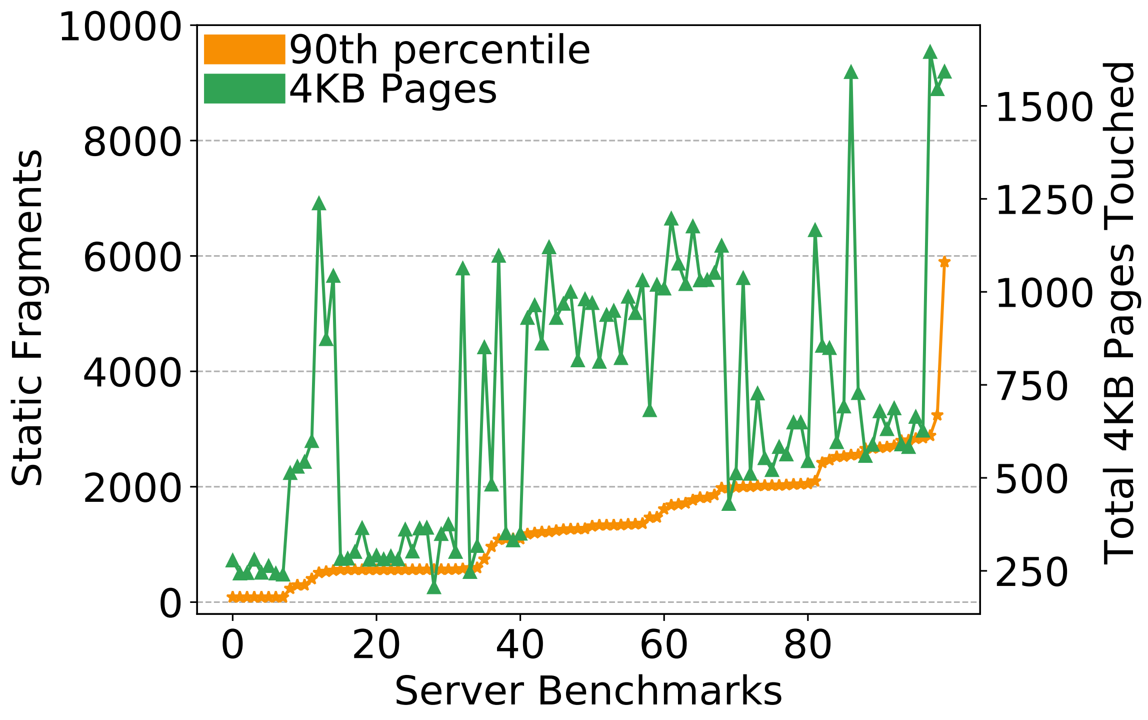


Figure 3.23: Static Fragments and Code Pages

Figure 3.23 presents the number of static fragments that account for 90% of program execution, as well as the number of static 4KB code pages touched by all benchmarks. We observe that in many cases, when the number of static fragments is higher, the number of 4KB pages touched is also relatively higher. Conversely, we also observe that for some benchmarks, although the number of fragments accounting for execution is small, the number of 4KB pages touched is quite high.

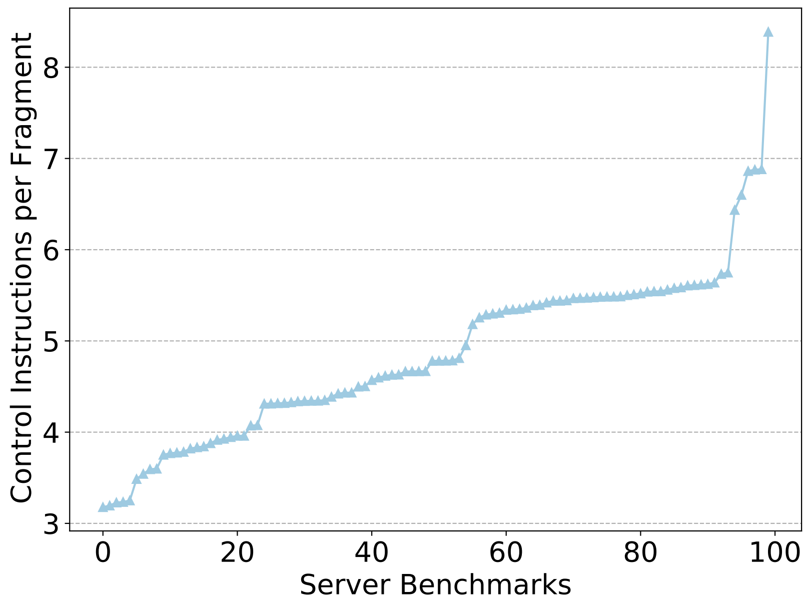


Figure 3.24: Control Instructions in Fragment

3.7.2 Control Instructions Enclosed in a Fragment

Figure 3.24 displays the average number of control instructions contained within a fragment for each benchmark studied. The results indicate that, on average, many benchmarks have between four and six control instructions within a fragment. In Table 3.1, we present the average number of control instructions within a fragment for every benchmark.

3.7.3 Fragment Level Control Flow

Lastly, we present observations regarding the successor fragments for the dynamic fragments illustrated in Figure 3.25. We categorize fragments into those with one successor fragment and those with two successor fragments.

When the successor fragment starts at the target of a return, we do not differentiate between successor fragments based on the return target. Such a fragment will have multiple successor fragments only if it also has another successor frag-

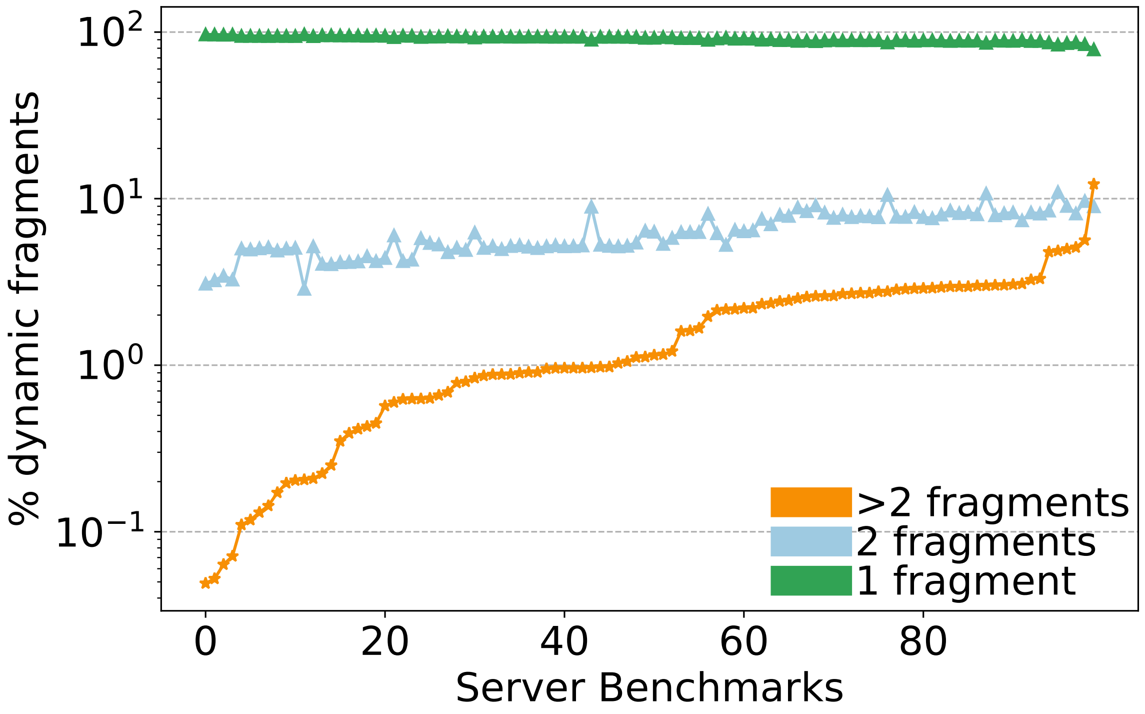


Figure 3.25: Fragment Level Control Flow Characteristics

ment starting at the target of a call instruction. Referring back to Figure 3.25, suppose function F1 is called from another call site besides call site A. In this scenario, the successor fragment for frag3 would be a different return fragment, not frag6. However, we still consider it to be a single successor fragment, because the exact identifier of the return fragment can be obtained using a return address stack.

Many benchmarks exhibit approximately 94% or higher of fragments with a unique successor fragment. About 4-5% of the fragments have two successor fragments, while some benchmarks show a higher percentage, ranging from 7-11%. The percentage of fragments with more than two successor fragments is close to zero for some benchmarks, while slightly higher for others, generally under 3% for most benchmarks. An additional observation regarding successor fragments is presented in Table 3.1.

App	L1i MPKI	% frag with 1 next frag	% frag with 2 next frags	% frag with >2 next frags	Avg CInstr per Frag	95th % frags	95th % funcs
secret121	3.2	94.9 (45.9,48.9)	5.0 (1.8,3.2)	0.1	4.8	98	38
secret141	3.2	94.9 (46.0,49.0)	5.0 (1.8,3.2)	0.1	4.8	99	38
public9	3.2	84.2 (37.7,46.4)	11.0 (4.0,7.0)	4.9	8.4	1048	237
secret100	3.2	94.8 (45.9,48.9)	5.0 (1.8,3.2)	0.1	4.8	106	39
secret103	3.3	94.8 (45.9,48.9)	5.1 (1.8,3.3)	0.1	4.8	120	42
public27	3.3	94.9 (46.0,49.0)	4.9 (1.8,3.1)	0.2	4.7	105	39
public30	3.4	94.7 (45.9,48.8)	5.1 (1.8,3.3)	0.2	4.7	125	43
public31	3.5	94.8 (45.9,48.9)	5.0 (1.8,3.2)	0.2	4.7	117	42
public29	3.6	94.6 (45.8,48.8)	5.2 (1.8,3.3)	0.2	4.7	136	46
public40	5.6	92.6 (43.9,48.6)	5.3 (3.7,1.6)	2.2	6.4	2641	493
server001	9.9	92.1 (44.1,48.0)	6.3 (3.3,3.0)	1.6	6.9	944	238
secret132	10.1	92.0 (45.2,46.8)	6.3 (3.2,3.1)	1.7	6.9	1016	252
secret137	10.2	92.1 (44.7,47.4)	6.3 (3.2,3.1)	1.6	6.9	984	246
public8	10.7	91.3 (43.9,47.4)	6.5 (3.4,3.1)	2.2	6.6	1248	305
public57	12.8	90.1 (40.5,49.6)	9.0 (5.5,3.4)	1.0	5.0	1832	375
server002	14.6	95.7 (47.5,48.3)	4.1 (0.2,3.8)	0.2	3.3	251	91
public11	16.0	86.3 (41.5,44.8)	10.7 (4.8,6.0)	3.0	5.7	2111	418
public54	16.5	89.9 (42.8,47.2)	8.1 (4.1,4.0)	2.0	5.5	2948	618
public1	16.8	86.7 (42.0,44.8)	10.5 (4.7,5.8)	2.8	5.6	2405	453
server003	19.0	78.8 (33.4,45.4)	9.0 (4.7,4.3)	12.2	4.8	8818	1231
server004	20.3	84.7 (40.8,43.9)	9.7 (4.8,4.9)	5.6	5.4	4714	888
public48	21.7	91.7 (43.2,48.5)	6.2 (4.1,2.1)	2.1	4.5	3488	755
public49	21.8	91.4 (43.0,48.4)	6.4 (4.2,2.2)	2.2	4.6	3342	736
secret128	22.4	88.3 (42.5,45.8)	9.1 (5.1,4.0)	2.6	5.5	2700	562
server009	23.0	86.0 (42.9,43.1)	9.0 (5.0,4.0)	5.0	5.5	2821	578
public55	23.3	90.6 (42.9,47.8)	7.0 (3.8,3.2)	2.4	5.7	3348	731

Table 3.1 continued from previous page

App	L1i MPKI	% frag with 1 next frag	% frag with 2 next frags	% frag with >2 next frags	Avg CInstr per Frag	95th % frags	95th % funcs
public41	23.6	90.1 (42.4,47.7)	7.6 (4.2,3.4)	2.3	5.6	3347	739
public51	24.0	93.5 (44.8,48.7)	5.4 (3.1,2.2)	1.2	5.3	3173	770
server010	24.2	88.6 (42.7,46.0)	8.9 (5.2,3.6)	2.5	5.5	2637	564
public68	24.3	88.8 (43.1,45.8)	8.3 (5.3,3.0)	2.9	5.6	2238	519
server011	24.8	86.7 (41.1,45.6)	8.5 (5.4,3.1)	4.8	5.4	4847	956
secret111	25.3	89.6 (43.6,46.0)	8.0 (5.5,2.5)	2.4	5.6	2006	495
secret131	25.3	89.7 (43.6,46.0)	7.9 (5.4,2.5)	2.4	5.6	2022	498
public67	25.5	89.5 (43.4,46.1)	7.7 (5.2,2.5)	2.8	5.4	1962	491
public6	25.6	89.4 (43.3,46.1)	7.8 (5.3,2.5)	2.8	5.5	1829	453
public7	25.6	89.5 (43.3,46.2)	7.8 (5.2,2.6)	2.7	5.3	2311	553
public3	25.6	89.0 (43.0,46.0)	8.0 (5.5,2.5)	2.9	5.5	1811	442
public65	25.7	89.4 (43.2,46.2)	7.8 (5.3,2.5)	2.9	5.4	1975	509
public66	25.7	89.4 (43.5,46.0)	7.9 (5.4,2.5)	2.7	5.5	1890	470
server012	25.8	89.0 (42.8,46.2)	8.4 (5.3,3.1)	2.6	5.3	2815	607
public46	25.8	92.5 (43.6,48.9)	6.3 (4.3,2.0)	1.1	3.9	2593	660
public10	25.8	88.6 (42.4,46.1)	8.5 (5.3,3.2)	3.0	5.3	2562	563
public69	26.0	86.8 (42.8,44.0)	8.1 (4.8,3.3)	5.1	5.2	3037	650
secret105	26.0	89.6 (43.3,46.3)	7.7 (5.4,2.4)	2.7	5.6	1991	512
public4	26.0	89.0 (43.1,45.9)	8.0 (5.4,2.5)	3.0	5.5	1839	471
public2	26.1	88.9 (42.9,46.0)	8.1 (5.5,2.6)	3.0	5.5	1897	475
secret120	26.1	89.7 (43.5,46.3)	7.7 (5.4,2.3)	2.6	5.6	1962	504
public50	26.1	93.0 (44.5,48.5)	5.8 (3.2,2.6)	1.2	5.5	3087	784
public64	26.1	89.5 (43.2,46.2)	7.6 (5.2,2.4)	2.9	5.5	1890	489
secret133	26.2	89.3 (43.3,46.0)	8.0 (5.6,2.4)	2.7	5.6	1910	472
public62	26.2	88.8 (42.9,45.9)	8.2 (5.7,2.5)	3.0	5.5	1876	463

Table 3.1 continued from previous page

App	L1i MPKI	% frag with 1 next frag	% frag with 2 next frags	% frag with >2 next frags	Avg CInstr per Frag	95th % frags	95th % funcs
public63	26.2	88.8 (42.9,45.9)	8.2 (5.6,2.6)	3.0	5.4	1999	493
public5	26.3	89.3 (43.1,46.2)	7.8 (5.3,2.5)	2.9	5.5	1961	507
public47	26.4	92.5 (43.6,48.8)	6.4 (4.3,2.1)	1.1	3.9	2568	658
public0	26.6	88.7 (42.7,45.9)	8.3 (5.4,2.9)	3.1	5.3	2817	589
public61	26.6	88.7 (42.7,46.0)	8.3 (5.2,3.1)	3.0	5.3	3063	636
secret12	26.7	88.6 (42.7,45.8)	8.1 (5.3,2.9)	3.3	5.3	3904	799
public60	26.8	88.5 (42.9,45.6)	8.3 (5.4,2.9)	3.3	5.4	2763	585
server013	26.8	89.1 (43.1,46.1)	8.2 (5.4,2.8)	2.6	5.4	3118	634
server014	28.8	95.5 (46.7,48.8)	4.2 (1.4,2.7)	0.3	3.2	315	114
server015	29.1	95.7 (46.9,48.8)	4.0 (1.4,2.7)	0.3	3.2	312	113
public53	33.2	89.5 (42.6,46.9)	7.4 (3.4,4.0)	3.1	4.6	3106	765
server016	36.6	93.4 (45.0,48.4)	5.4 (1.9,3.5)	1.1	5.5	3132	831
server017	43.2	94.1 (44.1,50.0)	5.1 (1.8,3.3)	0.9	4.6	605	239
server018	43.4	94.1 (44.0,50.1)	5.0 (1.8,3.2)	0.9	4.6	605	239
server019	44.4	94.0 (44.1,50.0)	5.1 (1.8,3.2)	0.9	4.6	606	240
public52	44.6	91.3 (44.0,47.4)	6.5 (2.9,3.6)	2.2	3.8	3186	807
public34	45.6	93.8 (45.5,48.3)	5.2 (1.9,3.3)	1.0	4.3	600	239
server020	46.0	93.9 (45.3,48.6)	5.2 (1.9,3.3)	0.9	4.5	606	240
public39	46.9	93.8 (45.5,48.3)	5.2 (1.9,3.3)	1.0	4.3	600	239
public76	46.9	93.8 (45.6,48.3)	5.2 (1.9,3.3)	1.0	4.3	600	239
public38	47.1	93.7 (45.5,48.3)	5.3 (1.9,3.4)	1.0	4.4	600	239
secret10	47.2	93.9 (45.6,48.3)	5.2 (1.9,3.3)	0.9	4.4	601	239
public36	47.3	93.9 (45.6,48.3)	5.2 (1.9,3.3)	1.0	4.3	600	239
server021	47.3	93.9 (45.5,48.3)	5.3 (1.9,3.3)	0.9	4.4	601	239
public75	48.1	93.8 (45.5,48.3)	5.3 (1.9,3.3)	1.0	4.3	600	239

Table 3.1 continued from previous page

App	L1i MPKI	% frag with 1 next frag	% frag with 2 next frags	% frag with >2 next frags	Avg CInstr per Frag	95th % frags	95th % funcs
public33	48.3	93.9 (45.6,48.3)	5.2 (1.9,3.3)	1.0	4.3	600	239
public37	48.3	93.7 (45.5,48.3)	5.2 (1.9,3.3)	1.1	4.3	602	240
server022	48.4	93.9 (45.6,48.3)	5.1 (1.9,3.3)	0.9	4.4	600	240
public35	48.6	93.8 (45.5,48.3)	5.2 (1.8,3.3)	1.0	4.3	601	240
server023	48.7	92.9 (44.8,48.1)	6.3 (2.2,4.0)	0.8	4.4	3151	834
server024	49.7	93.6 (45.3,48.3)	5.8 (2.0,3.8)	0.6	4.3	3125	833
server025	51.6	93.4 (45.4,47.9)	6.0 (1.9,4.1)	0.6	4.0	3041	767
server026	54.4	94.0 (45.7,48.4)	5.3 (1.7,3.6)	0.7	4.1	3096	794
server027	54.6	93.9 (45.6,48.4)	5.4 (1.8,3.6)	0.6	4.1	3102	795
server028	56.8	95.1 (46.3,48.7)	4.5 (1.5,3.0)	0.4	3.9	2176	684
server029	57.3	95.4 (46.6,48.8)	4.2 (1.3,2.8)	0.4	4.0	2175	685
server030	58.4	95.0 (46.4,48.7)	4.4 (1.4,3.0)	0.6	3.9	2212	692
server031	59.4	95.4 (46.6,48.8)	4.2 (1.3,2.9)	0.4	3.8	2176	686
secret113	60.0	95.3 (46.5,48.8)	4.2 (1.4,2.8)	0.4	3.8	2185	688
server032	62.9	96.5 (47.5,49.0)	3.4 (0.9,2.6)	0.1	3.5	2129	674
public44	63.8	95.1 (46.4,48.7)	4.3 (2.1,2.2)	0.6	3.2	2165	640
server033	64.9	94.1 (45.0,49.1)	5.1 (2.1,3.0)	0.8	3.8	1182	442
server034	65.1	94.3 (45.1,49.1)	4.9 (2.0,2.9)	0.8	3.8	1187	444
server035	66.6	94.5 (45.5,49.1)	4.8 (1.7,3.0)	0.7	3.8	1188	445
public45	68.2	95.2 (46.5,48.7)	4.2 (2.0,2.2)	0.6	3.2	2153	635
server036	74.8	96.9 (48.1,48.8)	2.9 (0.5,2.4)	0.2	3.8	428	176
server037	80.0	96.7 (47.7,48.9)	3.3 (0.9,2.3)	0.1	3.6	634	250
server038	80.1	96.7 (47.8,48.9)	3.2 (0.9,2.3)	0.1	3.6	631	249
server039	80.8	96.9 (48.1,48.7)	3.1 (0.4,2.6)	0.0	3.5	603	233

Table 3.1 continued from previous page

App	L1i MPKI	% frag with 1 next frag	% frag with 2 next frags	% frag with >2 next frags	Avg CInstr per Frag	95th % frags	95th % funcs

Table 3.1: Fragment Level Control Flow and L1i MPKI

Table 3.1 presents detailed benchmark data, including the L1i Misses Per Kilo Instructions (MPKI) for a 32KB, 8-way associative cache in data column 1 (L1i MPKI). Data column 2 (% frag with 1 next frag) shows the percentage of dynamic fragments with one successor fragment. Further it presents a two tuple of the number of fragments whose unique successor fragment is a call fragment and the number of fragments whose unique successor fragment is a return. When the successor fragment is a return, the successor fragment could be multiple return fragments, but we treat all of these fragments as one. Data column 3 (% frag with 2 next frags) shows the percentage with two successor fragments. Further it presents a two tuple of the number of fragments whose successor fragments are two call fragments or whose successor fragments are a call fragment and a return. Data column 4 (% frag with >2 next frags) shows the percentage with more than two successor fragments. Data column 5 (Avg CInstr per Frag) shows the average number of control instructions per fragment. Data column 6 (95% frags) shows the number of fragments that account for 95% of the program execution. Data column 7 (95% funcs) shows the number of functions that account for 95% of the program execution. The benchmarks exhibit L1i MPKI values ranging from 3.2 to 80.

We categorize the benchmarks into five bins based on their L1i MPKI values: 0-15, 15-30, 30-45, 45-60, and greater than 60. The first 16 benchmarks in Table 3.1 are part of bin 1, the next 45 benchmarks are part of bin 2, the following 6 benchmarks are part of bin 3, the subsequent 23 benchmarks are part of bin 4, and the final 10 benchmarks are part of bin 5. We observe that the percentage of dynamic fragments with more than two successor fragments is relatively high in the second bin (15-30 MPKI), with percentages as high as 12.2% for server003 and 5.1% for public69. These percentages are significantly lower (close to zero) for benchmarks in the higher MPKI bins, where code cache blocks are frequently moved between the lower-level and higher-level caches. Consequently, fragment-level control flow is easier to capture for applications with high MPKI, where there is more frequent between the cache levels.

From data column 7, we observe that the number of functions accounting for 95% is mostly under 1000 for most benchmarks.

We make three key observations regarding fragments and fragment-level control flow:

- A few thousand static fragments account for the majority of program execution, suggesting the feasibility of maintaining fragment-level information in reasonably sized tables.
- Dynamic fragments typically enclose an average of 4-6 control instructions, indicating that sequencing the program at the fragment level will likely go past those many control instructions at each step.
- Most dynamic fragments have a unique successor fragment, with a few having two successor fragments. Applications with higher L1i MPKI exhibit a higher percentage of fragments with a unique successor fragment.

A scheme that sequences high-level program control flow at the fragment level can potentially resolve upcoming high-level control flow with high accuracy by predominantly following a single path, as many fragments have a unique successor fragment. In cases where fragments have multiple successor fragments, the scheme can selectively follow a second path. Furthermore, the structures responsible for fragment-level tracking can be decoupled from the logic involved in fetching instructions, allowing the scheme to go past multiple control instructions in a single step. This suggests that a high-level control flow sequencing scheme could effectively determine the blocks of instructions and other closely related information necessary for processing of the instructions in the blocks.

3.7.4 Summarizing Potential for a Scheme

To summarize, for the commercial applications studied, we observe a very frequent movement of blocks of instructions between the lower-level cache and the higher-level cache. Additionally, for these applications, the data suggests that fragment-level information is straightforward to capture and that sequencing the program at the fragment level is likely to accurately resolve upcoming high-level control flow, going past multiple control instructions in each step. This process aids in determining the blocks of instructions likely to be referenced by the processor.

Building on these observations, this thesis presents ***Instruction Presending*** [56], a scheme that sequences fragment-level control flow independently of the logic for fetching instructions, thereby operating independently of BTBs, branch predictors, and the return address stack required to accurately determine the next program counter (PC). This scheme identifies the blocks of instructions, instruction TLB entries, and BTB entries likely to be needed by the processor and sends these blocks to the L1i, instruction TLB entries to the iTLB, and BTB entries to the L1 BTB in a just-in-time manner. Since this scheme operates independently of fetch logic, it uses identifiers and markers that are periodically validated to ensure alignment with the actual instructions fetched and executed by the processor. In the event of a divergence, the scheme restarts the process.

3.8 Summary

In this chapter, we studied the Instruction Supply problem and the associated challenges in an era where application code footprints overwhelm the size of primary structures, resulting in frequent code movement between cache levels. We reviewed the existing body of work aimed at tackling this challenge, focusing primarily on Instruction Prefetching schemes and providing a taxonomy for these schemes. Additionally, we discussed BTB and iTLB management schemes. Subsequently, we discussed an empirical potential for a scheme, Instruction Presending, discussing its place within this taxonomy and highlighting how it differs from other approaches. In the subsequent chapters, we will describe this scheme in detail, explaining how it facilitates the movement of code cache blocks and other related information, such as BTB and iTLB entries.

Chapter 4

Instruction Presending

Instruction Presending creates a shadow program representation, independent of data structures used for fetching instructions such as BTBs and branch predictors. This representation encodes information about fragments and the control flow across fragments. Once created, it is used to traverse the control flow at the fragment level, determining the cache blocks of instructions that the processor is likely to need and moving them to the L1i in a just-in-time manner.

In this chapter, we first provide a detailed description of the composition of the shadow program representation and the necessary hardware data structures used to capture this representation, along with associated issues and their handling. We then explain how these data structures are constructed. Next, we describe the usage of this representation (or the operation of the instruction presending scheme), along with associated issues and their handling. Finally, we discuss implementation aspects and alternatives, and make concluding remarks.

4.1 Program Representation

We begin by outlining the objectives of the program representation, followed by an in-depth examination of the various components of the program representation.

The key unit of program representation is the ***fragment***, as described in the

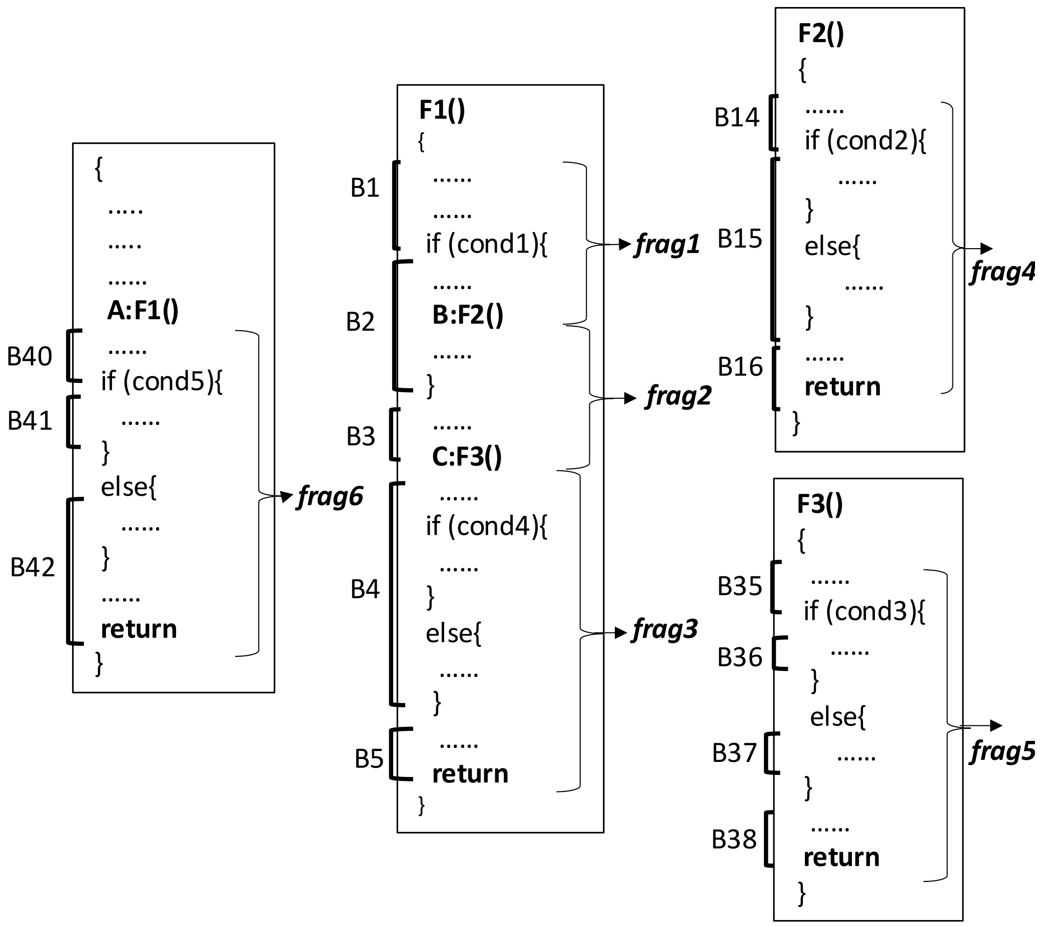


Figure 4.1: Fragments Example

previous chapter. To recap, fragment or a static fragment refers to a part of the program that starts at the target of a call and ends at another call or a return, representing a portion of the dynamic instruction stream delineated by calls and returns. A program can be partitioned into static fragments (similar to multiblocks [61]). Dynamic fragments are dynamic instances of these static fragments.

Here, we recap the example shown in the previous chapter: Figure 4.1 shows a function F1 which calls functions F2 and F3. The function F1 is called from call site A. The function F1 is delineated into three fragments:

- Fragment 1 starts at the beginning of the function and continues until the call to F2, composed of code cache blocks B1, B2 and B3 (B3 is accessed when cond1 evaluates to false).
- Fragment 2 starts at the return from F2 and continues until the call to f3, composed of code cache blocks B2 and B3.
- Fragment 3 starts at the return from f3 and continues until the return from F1, composed of code cache blocks B4 and B5. Furthermore, Fragment 3 contains local control flow, specifically an if-then-else construct.

Function F2 makes no additional function calls and thus constitutes a single fragment, Fragment 4, encompassing the entire function from start to return, composed of code cache blocks B14, B15, and B16. Similarly, function f3 makes no additional function calls and constitutes a single fragment, Fragment 5, composed of code cache blocks B35, B36, B37, and B38.

Following the return from F1, we have one fragment, Fragment 6, composed of code cache blocks B40, B41, and B42.

The control flow between these fragments is as follows:

- From Fragment 1 to Fragment 4 (could go from Fragment 1 to Fragment 5 if cond1 evaluates to false).
- From Fragment 4 to Fragment 2
- From Fragment 2 to Fragment 5
- From Fragment 5 to Fragment 3
- From Fragment 3 to Fragment 6

A program representation required to move blocks of instructions for this program snippet must include information about the blocks accessed by each of these fragments (Fragment 1 to Fragment 6) and the control flow from one fragment to the next. While these pieces of information are sufficient to determine the blocks of instructions the processor is likely to reference, they lack any notion of time. This is important to ensure that blocks are supplied in a timely manner to ensure the processor is not stalled waiting for blocks of instructions. The program representation can be augmented with additional information per fragment to aid in this process. We will describe this in more detail in Section 4.1.4.

More specifically, a program representation capable of traversing fragment-level control flow and correctly moving blocks of instructions for a program needs to:

- Correctly identify the fragments.
- Store information about the blocks accessed by different fragments.
- Maintain information about the successor fragments for each of these fragments.
- Maintain information about the amount of time it would take to execute the fragment.

This information collectively facilitates fragment-level sequencing coupled with the movement of code cache blocks.

Next, we provide a more detailed description of such a program representation. First, we describe the method for identifying fragments, followed by a description of the different components of the program representation.

4.1.1 Identifying Fragments

Fragments start at the target of a call or a return instructions. To recap, we refer to the fragment at the target of a call instruction as a ***call fragment***. (Note that the fragment starts at the target of the call instruction, not at the call instruction itself). We refer to the fragment at the target of a return instruction as a ***return fragment*** (Note that the fragment starts at the target of the return instruction, not at the return instruction itself). For example, Fragment 4 in Figure 4.1 is a call

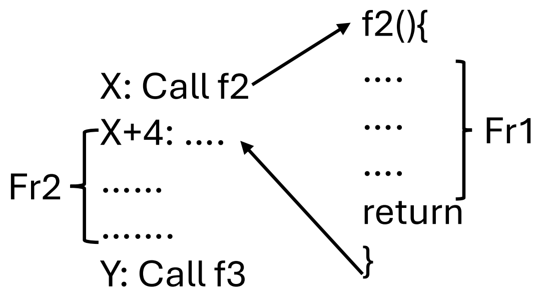


Figure 4.2: Identifying Program Fragments

fragment at the target of a call instruction to F2(). Fragment 2 in Figure 4.1 is a return fragment at the target of a return instruction from F2().

Key issues are to identify **call fragments** following a direct call and an indirect call and identification of a **return fragment**.

Direct Calls

A natural identifier for a call fragment following a direct call is the call instruction address (callPC), as it uniquely identifies the part of the program executing immediately after the call instruction. In the example shown in Figure 4.2, Fr1 is the fragment that executes following the call to function F2 by the instruction at address X. Therefore, X serves as the identifier for Fr1.

Returns

A return fragment executed following a return instruction can be identified by the target of the return instruction which is the continuation of a callPC, rather than by the return instruction itself, since a return can have multiple targets as a function can be called from multiple call sites. In the example shown in Figure 4.2, Fr2 is the fragment that executes following the return to address X+4. Therefore, X+4 serves as the identifier for Fr2.

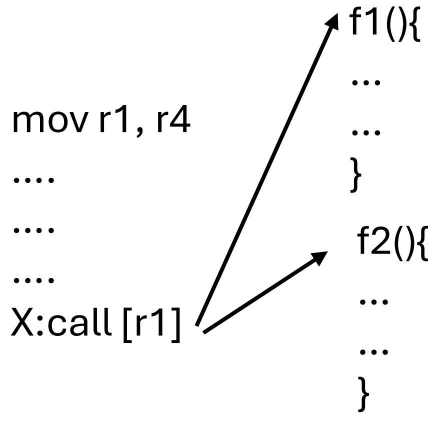


Figure 4.3: Indirect Calls Example

Indirect Calls

Indirect calls are instructions where the target instruction is not hardcoded in the instruction, but is determined at run time. They are a fundamental concept which allow programming languages to implement callbacks, polymorphism and dynamic function invocation [20], enabling more flexible software design. Typically, the callee function is determined by a value generated by an earlier instruction stored in a register. So the call instruction address (callPC) alone may not uniquely identify the call fragment, as it could refer to different fragments depending on the computed value. In such cases, a combination of the callPC and the target address of the call can be used collectively to identify the fragment.

For example, in Figure 4.3, based on the value of r1 computed earlier, the callee function could either be f1 or f2. The callPC X, in this context, would need to be associated with either f1 or f2 based on the target address (e.g., X+f1 or X+f2) to accurately identify the fragment following the call. This approach ensures that each fragment is uniquely identified even in scenarios involving indirect calls.

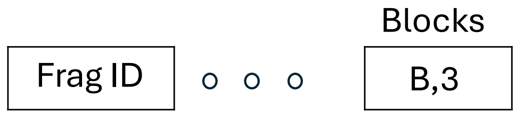


Figure 4.4: Code Region

4.1.2 Representing Code Cache Blocks in a Fragment

Unlike PC determination logic which requires knowing the next instruction to fetch, we are interested in the code cache blocks referenced by a fragment. This involves representing the code cache blocks accessed by every fragment in the table. In many cases, there is likely to be spatial locality in the blocks accessed by a fragment, resulting in accesses to contiguous blocks of code or **code regions**.

For instance, if a fragment accesses code cache blocks B, B+1, and B+2 sequentially, this sequence forms a code region of size 3 cache blocks starting at block B. This spatial locality allows us to compactly encode a code region accessed by a fragment, denoted as B,3 in Figure 4.4. This encoding indicates that the fragment accesses a code region starting at block B and spanning 3 consecutive cache blocks. We refer to the code region starting at the target of a call or a return as an **adjoint region**. An adjoint region can contain branch instructions within the same cache block that starts the fragment or in another block within the contiguous region, such as a loop branch where the loop spans multiple blocks.

Direct or indirect jumps can result in the access of discontiguous code regions. We refer to code regions discontiguous from the adjoint region as **separate regions**. One example that could result in separate regions is code packing optimization [27], which pack "hot code" together. Executing a "cold path" can lead to accessing discontiguous code cache blocks, resulting in the creation of separate regions.

In Figure 4.5, we observe a fragment starting at the beginning of function func, accessing code cache blocks B, B+1, and then making a direct jump to access block B+10. Although block B+10 is physically discontiguous from blocks B and B+1, it logically belongs to the same fragment, resulting in a separate region for the fragment.

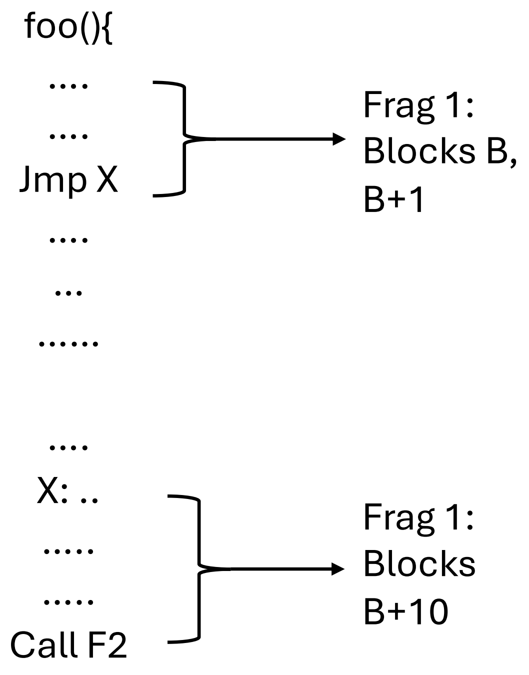


Figure 4.5: Discontiguous Region Example

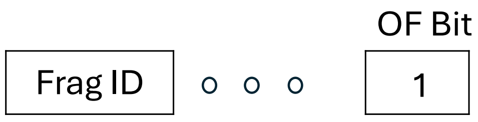


Figure 4.6: Setting OF Bit

To encode separate regions, a solution involves using an overflow bit (OF bit) to mark fragments that contain physically discontiguous blocks accessed. Figure 4.6 illustrates a fragment with its OF bit set to indicate that it accesses separate regions. This approach allows for the special handling of fragments that access separate regions while ensuring space efficiency in managing the common case of accesses to only adjoint regions for fragments. This is achieved by storing the separate regions only for fragments with their OF bit set in a separate structure. This separate structure can then be accessed only for fragments with the OF bit set, avoiding the need to maintain space for separate regions for all fragments.

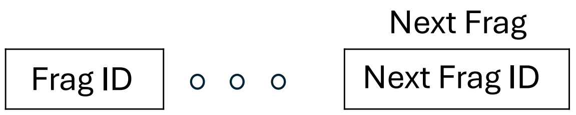


Figure 4.7: Successor Fragment After a Call

4.1.3 Representing Fragment Level Control Flow

Representing fragment-level control information is essential for managing high-level control flow sequencing in program execution. Control flow transfers between fragments primarily occur through call instructions (both direct and indirect) and return instructions.

- **Call Instructions** - When control flows from one fragment to another via a call instruction, the successor fragment identifier can be explicitly encoded. For example, in Figure 4.7, the fragment explicitly encodes the identifier of the successor fragment (Next Frag ID).

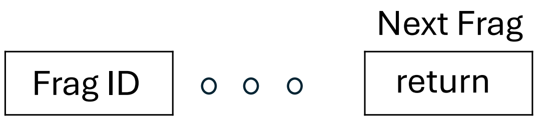


Figure 4.8: Successor Fragment After a Return

- **Return Instructions** - When control flows from one fragment to another via a a return instruction, successor fragment identifier may not uniquely identify the successor fragment because a fragment may have multiple call sites. Instead of explicitly encoding a unique successor fragment, the representation indicates that the control transfer occurs via a return.

Figure 4.8 illustrates this, where a fragment returns to its caller(s) without specifying a unique successor fragment identifier. The operation of the scheme can use a stack to obtain the exact fragment identifier of the successor fragment in this case, as we shall see in Section 4.5.

```
....
....
If (cond){
....
call f1();
}
....
call f2();
```

Figure 4.9: Multiple Successor Fragments Example

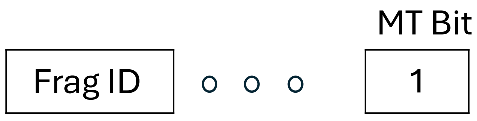


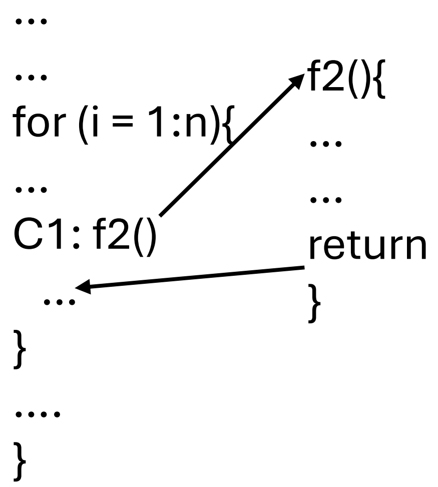
Figure 4.10: Setting MT Bit Example

Multiple successor fragments

In program execution, some fragments may have more than one possible successor fragment depending on run time conditions or indirect calls. For example, after evaluating a condition in fragment F1 (as shown in Figure 4.9), the successor fragment could either be the one corresponding to function call f1 or function call f2. Similarly, as shown in Figure 4.1, following frag1, the successor fragment could be either frag4 or frag5, depending on the evaluation of cond1.

To accommodate such scenarios, a fragment can be marked with an MT (Multiple Target) bit, as illustrated in Figure 4.10 where a fragment has its MT bit set. This bit indicates that the fragment has multiple potential successor fragments based on runtime conditions.

This approach allows for the special handling of fragments that have multiple successor fragments, while ensuring space efficiency in managing the common case of fragments with a single successor fragment (as shown earlier in Table 3.1, where a large percentage of fragments have a unique successor fragment). This is achieved by storing the alternate successor fragment only for fragments with their MT bit set in a separate structure. This separate structure can then be accessed only for fragments with the MT bit set, avoiding the need to maintain space for an alternate fragment for all fragments.



```
...
...                f2(){
for (i = 1:n){       ...
...                  ...
C1: f2()           return
  ...              }
}
....
}
```

Figure 4.11: Looping Fragments

Loops and Recursion

Loops and recursion are an example program constructs wherein a set of fragments is executed repeatedly. For instance, in Figure 4.11, function f2 is invoked from call site C1 within a for-loop scenario. This setup leads to the repetitive execution of both the call fragment f2 and the return fragment of C1 until the loop terminates after n iterations. To optimize efficiency, the scheme's operation must ensure that such fragments are not redundantly sequenced, especially considering that these blocks are likely already present in the L1i.

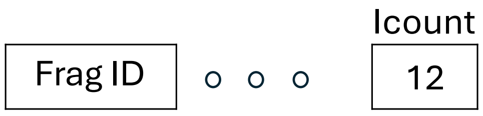


Figure 4.12: Fragment Instruction Counts

4.1.4 Representing Time to Execute a Fragment

All the components of the program representation described above collectively help determine the blocks the processor is likely to reference. However, this information alone is insufficient. Determining which blocks to supply is crucial, but a delayed supply can result in the processor waiting for an instruction for multiple cycles, leading to fetch stalls and pipeline bubbles. Conversely, supplying blocks too early can result in their premature eviction before they are referenced. Therefore, it is essential to have information that ensures the timely delivery of instruction blocks to the L1i.

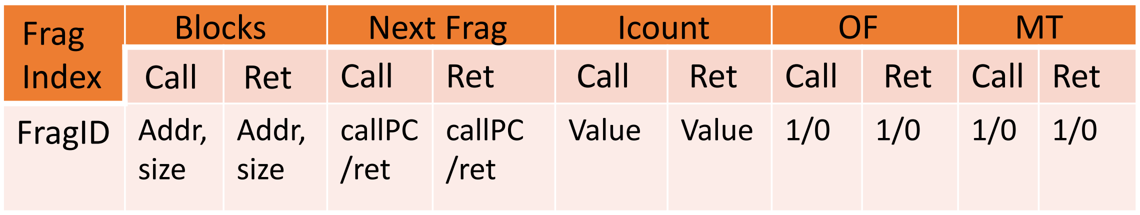
Timely delivery of instructions involves tolerating the latency of fetching instructions from a lower-level cache. While the representation identifies the fragments the processor is likely to fetch, additional information about the size of each fragment can facilitate a more timely supply of blocks to the L1i. Measuring the size of a fragment in dynamic instructions provides an estimate of the time the processor is likely to spend fetching instructions from a fragment before moving to another fragment. For this work, we use instruction counts to get an idea of time to execute a fragment. For example, in Figure 4.12, a fragment has its dynamic instruction count set to 12.

Alternatively, cycle counts can also be used for this purpose.

To mitigate variations in instruction counts across different executions of the fragment, the representation can track the minimum observed instruction count. This approach can sometimes result in staying slightly further ahead in the instruction stream, effectively tolerating the latency of fetching instructions from a lower-level cache.

4.2 Hardware Data Structures

This section integrates all components of the program representation and describes the hardware data structures utilized to store this representation.



Frag Index	Blocks		Next Frag		Icount		OF		MT	
	Call	Ret	Call	Ret	Call	Ret	Call	Ret	Call	Ret
FragID	Addr, size	Addr, size	callPC /ret	callPC /ret	Value	Value	1/0	1/0	1/0	1/0

Figure 4.13: Entry of a Fragment Table

4.2.1 Fragment Table

The primary data structure for storing information about fragments is the **Fragment Table** (FT). The program counter (PC) of a call instruction naturally identifies a fragment that begins at the target of the call, which is the entry point of a function, as we discussed in Section 4.1.1. When the called function returns, the target of the return instruction becomes the continuation code starting at the next PC (PC+4), which identifies the return fragment, as we discussed in Section 4.1.1. These two related fragments are combined into a single entry to maintain a compact representation. The two fragments are: (i) the **Call Fragment (Call)**, and (ii) the **Return Fragment (Ret)** associated with the call PC.

As illustrated in Figure 4.13, an entry in the Fragment Table contains the following components: (i) adjoint region for both the Call and Ret fragments, (ii) a count of dynamic instructions executed by the Call and Ret fragments, and (iii) the identifier of the fragment to be processed next for both the Call and Ret fragments.

The code region is stored as the first address of the block in the region and its size (in cache blocks). Two additional bits, the **Multi Target (MT) bit** and the **Overflow Regions (OF) bit**, for both the Call and Ret fragments, help accommodate fragments with multiple successor fragments and discontiguous fragments, as we had shown in Figures 4.10 and 4.6.

There are several alternatives for storing the address of the first block in the region:

– **Virtual Address**: Given that most lower-level caches are physical (where blocks would be moved from), using virtual addresses would necessitate a virtual to physical translation before blocks can be accessed and moved. Following the virtual to physical translation, the addresses of the remaining blocks in a code region, besides the first block, can be easily obtained by simple addition of offsets to the first block address.

– **Physical Address**: Using physical addresses to encode blocks accessed is the most natural choice since the lower-level caches are physical, and blocks would be moved from there. Moreover, all remaining block addresses in the code region, besides the first block, can be easily obtained by simple addition of offsets to the first block address. The usage of physical addresses could result in a separate region when the region (adjoint or separate) crosses a page boundary.

– **Cache Block Pointers**: Given our intention to use the program representation to move blocks from a lower-level cache to the L1i, a physical address can be replaced with a cache block pointer to the block in the lower-level cache as a space or storage optimization. The tradeoff here is that computing the addresses of the other blocks in the region besides the first block would require a tag lookup to obtain the complete block address of the first block. Using the block address obtained from the tag lookup for the first block, the addresses of the other blocks can be computed, similar to the usage of physical addresses.

4.2.2 Handling Indirect Calls/Returns - Modified Tagging

For direct calls, the callPC serves as a unique identifier for a fragment. However, for an indirect call, there could be multiple targets for the same callPC. Similarly, for a return, there could be multiple targets because a function could have multiple call sites.

To differentiate fragments corresponding to different call targets following an indirect call, we use a simple hash (sum or XOR) of the callPC and the target, as described in Section 4.1.1. The program representation remains agnostic to whether the call was direct or indirect, operating with fragment identifiers.

Return fragments are identified by the target of the return, which is callPC+4. When the successor fragment for a given fragment is a return, the next fragment identifier is simply encoded as a constant. During the operation of the scheme, a stack is consulted to obtain the exact fragment identifier in the case of a return fragment.

	Blocks	Blocks
Frag-ID	B1,3	B2,3

Figure 4.14: ORT Entry

4.2.3 Handling Discontiguous Fragments - Overflow Regions Table (ORT)

Many fragments have only an adjoint region that can be stored in the main Fragment Table (FT). However, several fragments access non-contiguous blocks, leading to access of separate regions. To accommodate these separate regions, we use an **Overflow Regions Table** (**ORT**). The ORT, similar to the FT, stores a code region as a block address and a size (number of contiguous blocks following the block address).

To handle a skewed distribution in the number of separate regions for a fragment, we have ORTs of different sizes, accommodating fragments with varying numbers of separate regions. For instance, an ORT-2 table entry can store two separate regions. Figure 4.14 illustrates an example ORT-2 entry storing two separate regions for a fragment. Similarly, an ORT-16 table entry can store up to sixteen separate regions for a fragment.

Accessing the ORT

The ORT is tagged separately and accessed if the Overflow (OF) bit is set in the Fragment Table (FT). Alternatively, the FT could store a pointer to the ORT entry for such fragments. However, this approach may lead to clutter in the FT, as each entry would require storage for a pointer, even when it is not necessary.

	Next Frag	Aging Bits	Aging Bits
Frag-ID	F3	2	2

Figure 4.15: DTT Entry

```
....
....
If (cond-highly biased not taken){
....
call f1();
}
....
call f2();
```

Figure 4.16: Multiple Successor Fragments With Biased Branch

4.2.4 Handling Multiple successor fragments - DTT/MTT

Some fragments have more than one possible successor fragment, as described in Section 4.1.3. To avoid clutter in the main Fragment Table (FT), which holds only a single target per fragment (since most fragments have a unique successor), we store additional targets in a separate table. The **Dual Target Table (DTT)** stores the identity of an alternate fragment target. Alternatively, a **Multi-Target Table (MTT)** can store the identity of multiple fragment targets.

In the example shown in Figure 4.16, after evaluating a condition, the successor fragment could be either the call fragment following the function call to f1 or the call fragment following the function call to f2. If the branch evaluates to true a few times and then subsequently evaluates to false (due to being highly biased towards not being taken), the successor fragment would eventually only be the fragment corresponding to the function call to f2. Capturing this information would require some bits for both paths, to indicate which of the successor fragments are active. For this reason, we have found it useful to maintain additional bits, called **aging bits**.

Figure 4.15 shows an example of a DTT entry storing an alternate fragment target for a fragment as F3, with the age of both successor fragments (fragment targets) set to 2. The aging bits for both paths are maintained in the DTT.

While we could store more than two targets in a Multi-Target Table, for most benchmarks considered in this work, two targets have been sufficient.

Accessing the DTT

The DTT is tagged separately and accessed if the MT bit is set in the Fragment Table. Alternatively, the Fragment Table could store a pointer to the DTT entry for such fragments, though this approach would lead to clutter in the Fragment Table, as each entry would require storage for a pointer, even when it is not necessary.

4.3 Constructing Tables

We describe the construction and population of the various data structures (FT/ORT/DTT) using Figure 4.18, in relation to the execution of the program snippet shown in Figure 4.17 that also highlights the code cache blocks to which the lines of the static program belong.

When the processor executes a call at Program Counter (PC) A, it marks the start of a new fragment A. As the processor accesses blocks B1, B2, and B10, which are part of this fragment, the block addresses are stored in a buffer. Although B10 is not shown as part of the fragment in Figure 4.17, it results from a jump within block B2 that transfers control to block B10 (not shown in the figure). Block B10 has a jump that transfers control back to block B2 (not shown in the figure). This could be due to code layout optimizations

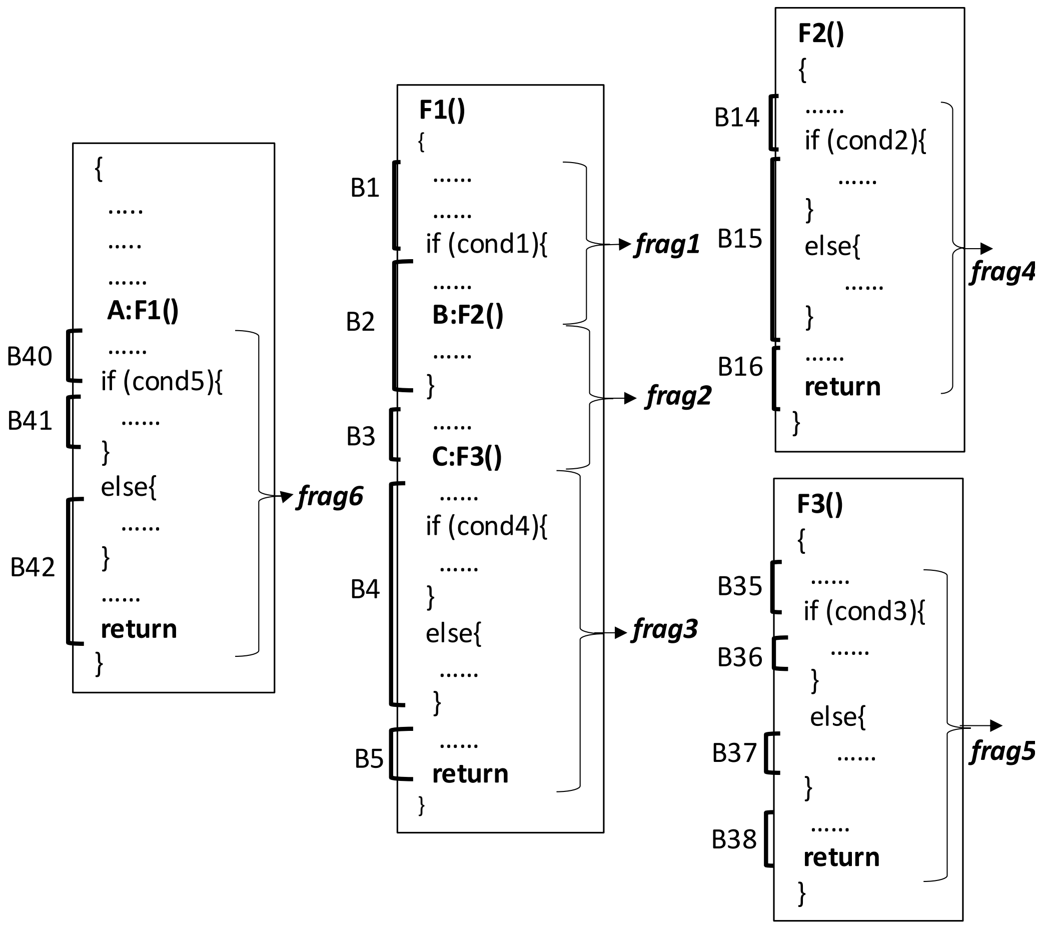


Figure 4.17: Example Program Snippet

that aim to pack hot code together and separate cold code, for example. Assuming the condition evaluates to true initially and the processor executes a call at PC B, this begins a new fragment B. The next fragment for fragment A is now fragment B, which is buffered by the processor. Additionally, the processor buffers the dynamic count of instructions executed as part of fragment A. When the call at PC A retires, a new entry for fragment A is created in the Fragment Table (FT). When the call at PC B retires, the fields for fragment A, including the dynamic count of instructions executed (12), the blocks ac-

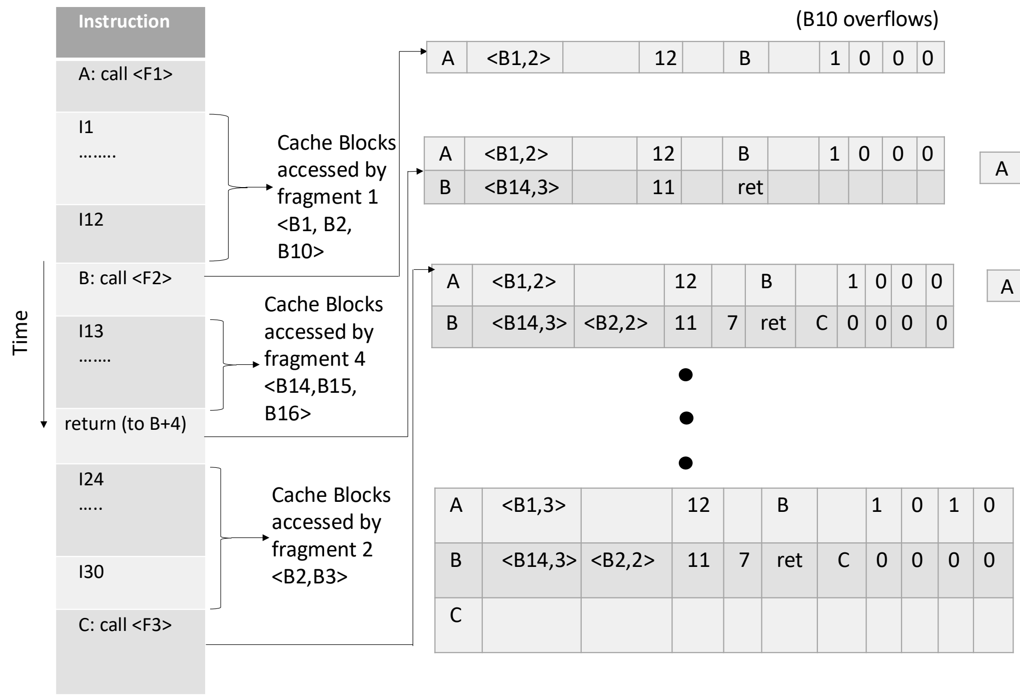

Instruction
A: call <F1>
I1
I12
B: call <F2>
I13
return (to B+4)
I24
I30
C: call <F3>
Time
Cache Blocks accessed by fragment 1 <B1, B2, B10>
Cache Blocks accessed by fragment 4 <B14,B15, B16>
Cache Blocks accessed by fragment 2 <B2,B3>
(B10 overflows)
ORT
A <B10,1>
DTT
A C 2 2

Figure 4.18: Fragment Table Construction Example

cessed (B1, B2, and B10), and the successor fragment (B), are stored in the FT. Additionally, a new entry for fragment B is created.

One important detail to note is that B1 and B2 are contiguous in the memory address space, while B10 is not; thus, fragment A is composed of one adjoint region and a separate region. In the main FT, the code region <B1,2> is stored for B1 and B2, and an ORT bit is set, indicating that the fragment has separate regions. The second region for fragment A <B10,1> is stored in the ORT.

As part of fragment B, the processor accesses blocks B14, B15, and B16 before executing a return. When the return following the call at PC B retires, the FT entry for fragment B is populated with the region of memory accessed <B14,3>, the dynamic instruction count (11), and the next fragment set as return. Following the return, the processor continues accessing instructions starting at B+4 and onward, which constitutes the return fragment (Ret) of B. Blocks accessed as part of the Ret fragment of B includes B2 and B3. Upon the retirement of the call at PC C, the contents of the Ret fragment of B are stored, with the code region set as <B2,2>, the count of retired instructions set to 7, and the next fragment set to C.

At a later point in the program's execution, if the condition following the call at PC A evaluates to false, then the next fragment for call fragment A would be C, not B (as previously stored). This would result in an IPU redirect and an update to the table. The FT entry for A would now have the DTT bit set, indicating multiple successor fragments. An entry for A would also be created in the Dual Target Table (DTT) with the alternate fragment target and the aging bits set to the maximum value for both possible fragment targets. Further block B3 would also be accessed as part of frag1 and would miss. This would update the adjoint region for the FT entry at A from <B1,2> to <B1,3>.

If an entry for a fragment already exists in the FT, the processor has no way of knowing to further update the FT entry. However, if the processor encounters a new cache miss for a fragment with an existing entry, this can trigger an update to the FT (or ORT/DTT) entry already present. In the absence of misses or redirects, the assumption is that the FT/ORT/DTT contains all necessary information to send the required blocks of a fragment to the L1i.

Alternatively, these tables could also be populated by software. This aspect is not explored in this dissertation.

4.4 Design Overview

The Instruction Presending Unit (IPU), illustrated in Figure 4.19, utilizes this learned program representation to send blocks of code within each fragment to the L1i as needed, proceeding to the subsequent fragment accordingly. The processor sends fragment identifiers to the IPU to indicate which frag-

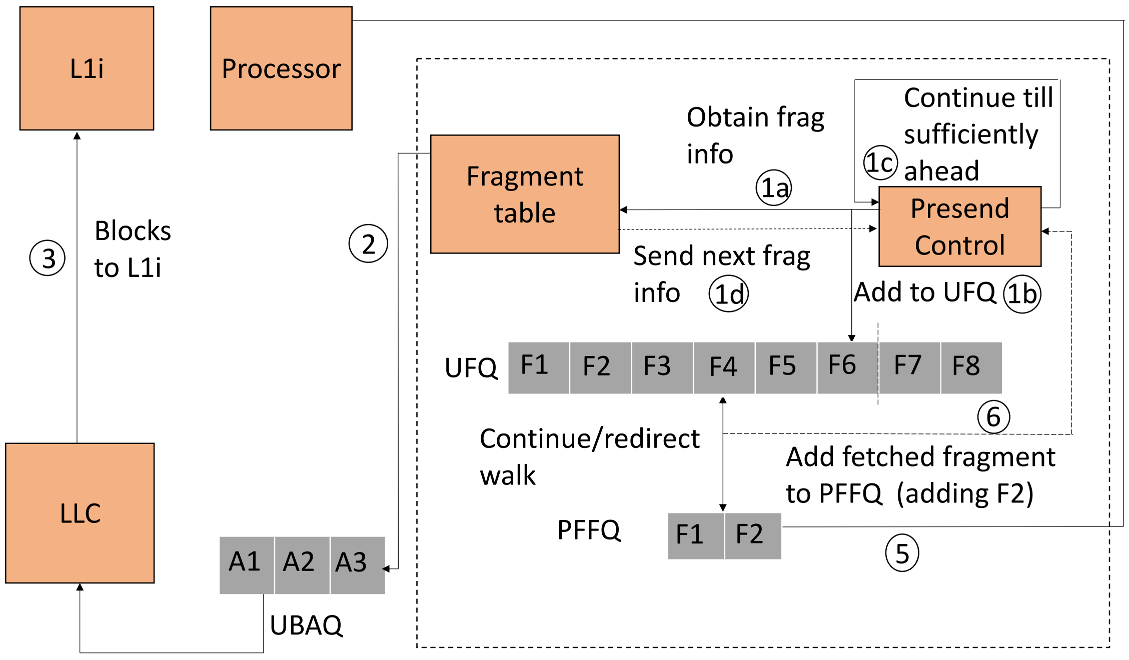


Figure 4.19: Overview of the Instruction Presending Unit (IPU)

ment is currently being processed. This communication enables the IPU to stay sufficiently ahead of the processor and to take corrective action if necessary, such as in cases of divergence. The operation of the IPU is detailed further in the following section.

Before describing the operation of the IPU, we briefly describe some other data structures that facilitate its operation:

- **Upcoming Fragments Queue (UFQ)** - This queue contains the fragment identifiers for the fragments that have been sequenced by the IPU. They are used to synchronize with the processor and keep sufficiently ahead.
- **Processor Fetch Fragments Queue (PFFQ)** - This queue contains the fragment identifiers for the fragments that have been fetched by the pro-

cessor. They are used to aid the IPU in staying sychronized with the processor.

– **Upcoming Block Addresses Queue (UBAQ)** - This queue contains the set of block addresses identified the IPU that are likely to be referenced by the processor. These addresses are used to move the corresponding cache blocks to the L1i based on the conditions described in Sections 4.5.2 and 4.5.3.

– **IPU Stack (IPUS)** - A stack used to traverse fragment-level control across returns, ensuring the correct sequencing of fragments following return instructions.

4.5 Using Program Representation – IPU Operation

Algorithm 1: Operation of the IPU

```
1  Next_Frag = Starting Fragment from Processor
2  while not sufficiently ahead (Step 1c) do
3      if Next_Frag = Call then
4          Access FT (Step 1a)
5          Push Return Fragment in IPUS
6          FragInfo = FragInfo from FT (Step 1d)
7          Update Next_Frag from FragInfo (Step 1d)
8      else
9          FragInfo = Pop from IPUS
10         Update Next_Frag from FragInfo (Step 1d)
11     Use FragInfo to update UFQ (Step 1b)
12     Add Blocks from FT to UBAQ (Step 2)
```

4.5.1 Fragment-Level Sequencing Coupled with Determination of Blocks

First, we describe the component of the scheme that employs high-level (fragment) sequencing to move code blocks from a lower-level cache to the L1i, ensuring that the cache is supplied with blocks the CPU is likely to reference soon. Additionally, we address issues related to the efficient movement of code blocks, focusing on transferring only active code blocks while avoiding the movement of blocks already present in the L1i.

Single Next Fragment

We describe the operation of the IPU, as depicted in Algorithm 1, for scenarios where fragments have a single successor fragment. This algorithm also references steps from Figure 1. The IPU functions to keep sufficiently ahead of the processor throughout program execution, operating with fragment identifiers.

When the Fragment ID corresponds to a call (step 1a), the IPU consults a table to retrieve information about the fragment, such as the blocks accessed by the fragment, the count of instructions within the fragment, and the next fragment ID. Additionally, it pushes information about the return fragment into the **IPU stack (IPUS)**. If the next fragment ID is a return, an entry is popped from the IPUS to obtain information about the next fragment target.

The fragment ID is stored in an Upcoming Fragments Queue (UFQ) (step 1b), and the blocks accessed by the fragment, obtained from the Fragment Table (FT) and occasionally from the ORT, are pushed into an Upcoming Block Addresses Queue (UBAQ) (step 2). This process is repeated for the next Fragment ID (step 1d) until the IPU is sufficiently ahead of the processor (step 1c).

After adding blocks to the UBAQ, a decision must be made regarding whether a block needs to be sent to the L1i, which is detailed in Section 4.5.2 and Section 4.5.3. Based on this decision, the actual blocks of instructions are placed in a Presend Block Queue (PBQ) (step 3), from where they proceed either to the L1i or directly to the processor (step 4).

Multiple Next Fragments

In cases where a fragment can have multiple successor fragments (indicated by the MT bit being set in the FT entry), the IPU has several options. The IPU could either stop, go down one of the two paths, or go down both paths. If the IPU is to explore multiple paths, it requires separate tracks (and thus separate stacks) to follow these different paths. Upon encountering such a fragment, the IPU stack (IPUS) is duplicated, and both paths are explored with their respective IPUS. The fragment ID for the alternate path is obtained from the Dual Target Table (DTT), which is used to pursue the second path. While there may be two paths, the IPU may choose to explore only one if the other path is inactive (indicated by its aging bits being set to 0).

Depending on the path taken by the processor (known via processor updates made to the PFFQ), the IPU can subsequently go down only one of the two forked paths. Additionally, aging bits for the path are also updated (the next fragment processor went down has its aging bits updated positively, while the other path is updated negatively, to ensure the IPU does not explore inactive or less frequent paths). For most benchmarks studied here, we see that exploring two paths suffices, for a few benchmarks exploring additional paths provides benefit.

4.5.2 Handling Cold Code Blocks

The IPU places block addresses in the Upcoming Block Addresses Queue (UBAQ). However, not all blocks are likely to result in a useful supply of instruction blocks to the L1i. Given that the IPU operates in a remote and decoupled manner, it needs to discern whether a block belongs to an inactive code region and is therefore unlikely to be referenced by the processor.

Studies have shown that server applications often contain parts of the program that are never referenced (cold code) [14]. Cold code is never loaded into the L2/LLC cache, so it is never added by the IPU to the UBAQ. However, there are program sections (blocks of instructions) that are likely referenced only once and then rarely, due to highly biased conditional branches, for example. While a region may be active, some blocks within it may become cold while others remain hot. Sending both types of blocks to the L1i can result in a wasteful supply of instruction blocks. To mitigate this, we can associate temperature bits (chosen to be 3 bits in this case) with each block. A low value indicates the block is cold, while a high value indicates it is hot.

The temperature of a block can be learned dynamically. Initially, all blocks can be set to a mid-temperature value (4, using 3 bits). When blocks are sent to the L1i, their temperature bits are incremented (up to a maximum value of 7) if accessed before eviction or decremented (down to a minimum value of 0) if not accessed. A block with a minimum temperature value is considered cold, and the decision to send blocks can be based on whether a block is non-cold.

This scheme requires maintaining temperature bits per block within the L2/LLC cache. Accessing these temperature bits would involve accessing the L2/LLC cache tags.

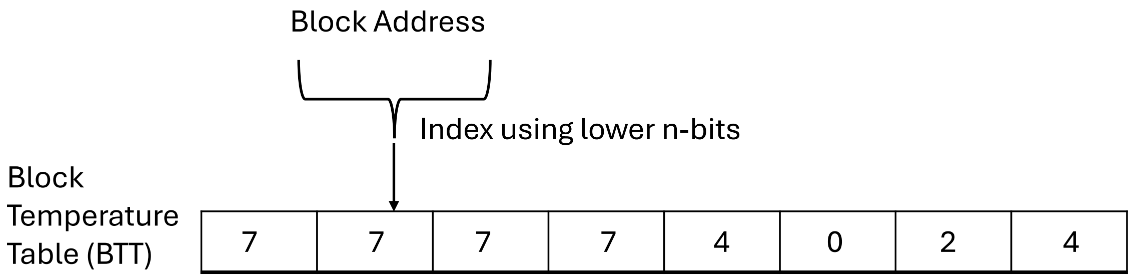


Figure 4.20: Block Temperature Table

To avoid this, we maintain a separate table called the **Block Temperature**

Table (BTT), accessed using the lower n-bits of the block number, as shown in Figure 4.20. The BTT is a linear array of counts indicating the temperature, with values ranging from 0 (cold) to 7 (hot). The BTT is consulted for block addresses in the UBAQ and only blocks with an adequate temperature are sent to the L1i.

4.5.3 Handling Blocks Already Present in the L1i

Another crucial aspect of the efficient operation of the IPU is determining whether a block is already present in the L1i.

To address this, prefetchers typically probe the L1i tags to verify the presence of a block. The IPU could employ a similar approach. However, this significantly increases the number of L1i tag probes and may present implementation challenges, as it would require additional read ports in the cache.

Alternatively, the IPU (or a prefetcher) could maintain a pseudo-inclusion bit in the L2/LLC for each block to track its presence in the L1i. The pseudo-inclusion bit would be updated as blocks are inserted or evicted from the L1i. These bits could be maintained alongside the L2/LLC tags. However, this approach also comes with the challenge of having to probe the L2/LLC tags to check for the presence of blocks in the L1i.

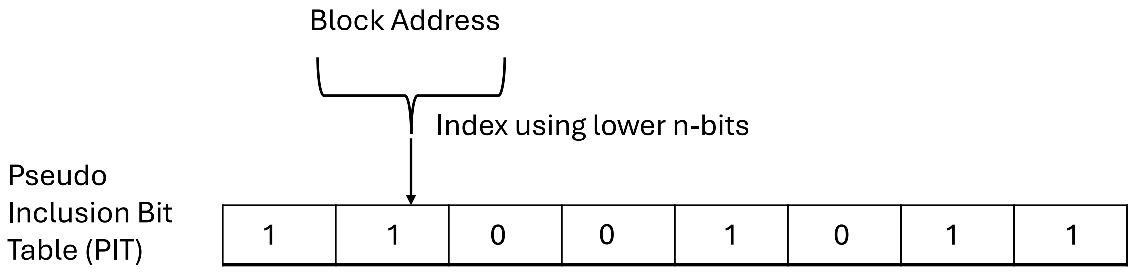


Figure 4.21: Pseudo Inclusion Bit Table

Maintaining these bits in a separate table, the **Pseudo Inclusion Bit Table (PIT)**, prevents unnecessary accesses to the L2/LLC tags. The PIT is a linear array of bits accessed using the lower n-bits of the block number, similar to the BTT, as illustrated in Figure 4.21. Our simulation results indicate that using a PIT instead of probing the L1i tags yields nearly identical results. Employing a good hashing scheme is likely to result in similar performance to maintaining a per-block inclusion bit in a lower-level cache. The only instances when a pseudo-inclusion bit leads to an incorrect block-sending decision are when the block is present in the cache at the time of decision-making but is evicted before being referenced, which occurs very infrequently.

We have examined both alternatives in the following chapter and present results using a PIT maintained separately from the L2/LLC tags.

Another alternative, which we have not studied in this thesis, is to maintain shadow L1i tags (SL1) along with replacement/reference bits. These can be updated using the block reference stream generated by the IPU and fed to the UBAQ. This approach allows the system to operate independently of L1i accesses. Any incoherence between the L1 tags and the SL1 tags can be corrected during block replacements from the L1i and on misses in the L1i, without impacting correctness.

4.5.4 Synchronizing High-Level Sequencing with the processor

Given that the key component for effectively supplying instruction blocks is the high-level sequencing facilitated by the learned program representation, the IPU must ensure that it stays synchronized with the processor. We describe this aspect of the scheme next.

The IPU tries to remain on the correct execution path. To ensure it is on track, as the processor fetches new fragments, the Fragment IDs are added to a Processor Fetch Fragment Queue (PFFQ). These Fragment IDs could be added during commit, and we describe the implications in Section 4.5.8. This allows the IPU to monitor the processor's current position in the program execution.

As new fragments are added by the IPU to the PFFQ, the processor compares the contents of the PFFQ with the Upcoming Fragments Queue (UFQ). This comparison allows the processor to verify whether the IPU is on track and to detect any divergence from the expected execution path. We next illustrate this with an example.

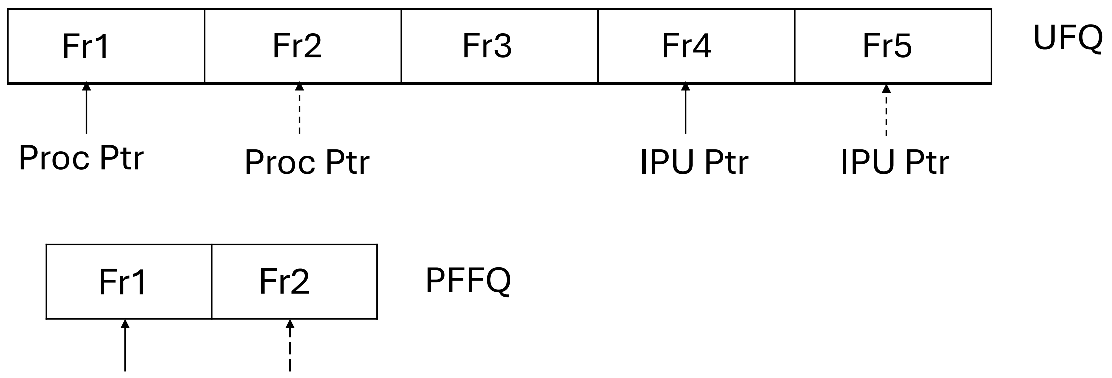


Figure 4.22: Synchronizing with the Processor

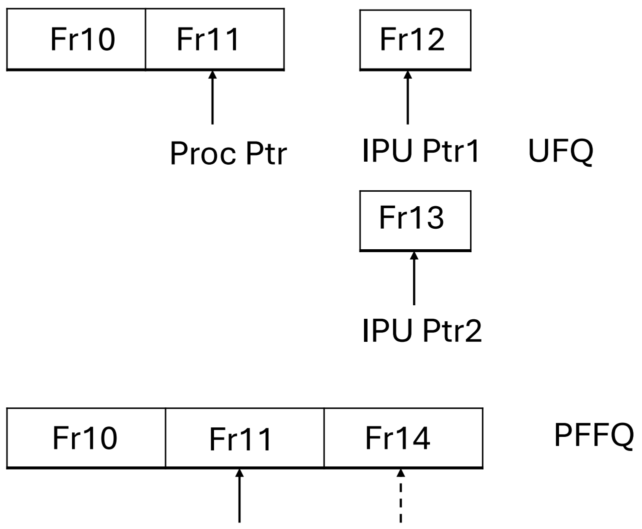


Figure 4.23: IPU Divergence from the Processor

Figure 4.22 illustrates how the IPU stays synchronized with the processor.

The IPU maintains a pointer to the fragment last added to the PFFQ (Proc Ptr) and a pointer to the last fragment sequenced (IPU Ptr) in the UFQ. Initially, the processor inserts Fr1 into the PFFQ. In the UFQ, the IPU's Proc Ptr points to Fr1, which was sequenced in the past, and the current IPU Ptr is at Fr4, the last fragment sequenced by the IPU. When the processor inserts Fr2 into the PFFQ, the Proc Ptr for the processor fragments is moved by one position in the UFQ. There is a match, indicating that Fr2 was sequenced correctly by the IPU. The IPU then continues to sequence additional fragments, moving the IPU Ptr to Fr5 in the UFQ.

There are instances where the processor may diverge from the fragments sequenced by the IPU. This occurs when the fragment obtained from the PFFQ does not match the fragment obtained from the UFQ. In such cases, the IPU experiences a *redirect* and begins sequencing from the fragment obtained from the PFFQ that was not present in the UFQ.

We illustrate the occurrence of divergence with an example, as shown in Figure 4.23. Following Fr11, let us assume there are three possible successor fragments: Fr12, Fr13, and Fr14. We store up to two successor fragments for a given fragment, so in this example, let us assume the successor fragments stored are Fr12 and Fr13. Following Fr11, the IPU sequences Fr12 and Fr13, resulting in two IPU pointers pointing to Fr12 and Fr13. However, when the processor inserts Fr14 into the PFFQ, this results in a divergence or mismatch when the Proc Ptr is moved in the UFQ, as Fr14 was not sequenced by the IPU and would not be present in the UFQ. This triggers a redirect within the IPU starting from fragment Fr14.

4.5.5 Keeping Sufficiently Ahead

In addition to supplying a set of code cache blocks that are likely to be referenced by the CPU, timeliness in the delivery of these blocks is crucial to avoid stalls while fetching instructions from the L1i. We next describe how the IPU maintains a sufficient lookahead to keep the processor supplied with blocks of instructions in a timely manner.

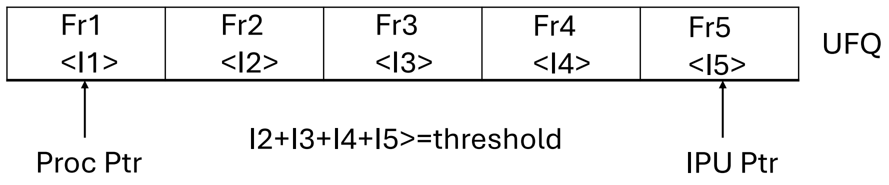


Figure 4.24: IPU Lookahead

Knowing the processor's current fragment allows the IPU to stay sufficiently ahead. This is achieved because the IPU has access to the instruction count information for the additional fragments present in the Upcoming Fragments Queue (UFQ), as the Fragment Table (FT) stores dynamic instruction counts for all fragments. This information enables the IPU to determine how far ahead it is from the processor's execution path. We describe this with an example.

Figure 4.24 shows the fragments sequenced by the IPU in the UFQ. The IPU has sequenced fragments Fr1 through Fr5 with instruction counts of I1 through I5. The processor pointer (Proc Ptr) in the UFQ is currently at Fr1. Let us assume there is a lookahead threshold that the IPU seeks to maintain. The IPU sequences fragments to ensure that the cumulative instruction count of fragments following Fr1 exceeds this threshold. In this case, sequencing up to fragment Fr5 ensures this criterion is met.

Next we describe how this lookahead threshold can be chosen. IPU seeks to keep sufficiently ahead to tolerate the latency of fetching instructions from the lower-level cache. In this regard, suppose the latency of fetching instructions from a lower-level cache is 20 cycles, and the fetch width of the machine is 8 instructions, IPU would seek to keep 20x8=160 instructions ahead to tolerate this latency. In program phases where the processor fetches fewer than 8 instructions, IPU is likely to keep slightly ahead.

Alternatively, one could use cycle counts in place of instruction counts. In this case, tolerating a fetch latency of 20 cycles would involve setting a lookahead threshold of 20 cycles.

4.5.6 Handling Loops and Recursion

Loops and recursion are program constructs where the same set of fragments are executed repeatedly until control flows to the continuation of the loop or the recursive call. However, the IPU does not need to repetitively send the same code to the L1i, as it is likely already present in the L1i. The IPU only needs to send blocks along the continuation path (alternate path) to stay sufficiently ahead and wait for the processor to reach the continuation of the loop or the recursive call.

This is monitored via the Processor Fetch Fragment Queue (PFFQ), which filters the insertion of the same fragment identifiers occurring within a short window (2-4 fragments) in the queue. This filtering mechanism allows the IPU to keep sufficiently ahead along the continuation path and continue sequencing more fragments along the continuation path only when the processor exits loop or recursion and inserts new fragment identifiers into the PFFQ. Such an operation allows the IPU to function effectively, remaining agnostic to loops and recursion within the program.

4.5.7 Processor Wrong-Path Execution

If the processor inserts fragment IDs into the PFFQ at fetch, there is a possibility of incorrect fragment IDs being inserted when the processor is fetching on an incorrect branch path, leading to unnecessary redirects of the IPU. However, this is a very infrequent occurrence due to the high-level sequencing employed.

First, for many fragments, there is a single successor fragment, and regardless of the branch outcomes within the fragment, the next fragment inserted into the PFFQ remains the same. Similar observations have been made by other studies exploiting control independence [7, 25, 61, 73].

Second, for fragments with multiple successors, we record two successors in the FT/DTT, and the IPU proceeds down both paths. Consequently, the IPU will only be incorrectly redirected if the processor's execution on an incorrect branch path leads to a successor fragment that is not one of the two fragments recorded in the FT/DTT that the IPU is pursuing. This occurs very infrequently, as is quantified in the next chapter in Section 5.10.

Therefore, the IPU can continue to operate using processor fetch fragment IDs with minimal redirects, even in the presence of wrong-path execution.

It is important to note that wrong-path execution does not affect the training of the FT/DTT, as they are created using the retired instruction stream.

4.5.8 Using Processor Commit Fragment Identifiers

While we have described using fetch fragment identifiers to synchronize the processor with the IPU, using commit fragment identifiers for this purpose is also an option. This approach has the advantage of not being affected by wrong-path execution, as discussed in Section 5.10, unlike fetch fragment identifiers. However, commit fragment identifiers are known much later in the instruction pipeline compared to fetch fragment identifiers.

When fragment-level sequencing experiences frequent redirects, using fetch fragment identifiers helps correctly redirect the IPU early, bringing it back on the correct path. In contrast, delay in correcting the IPU using commit fragment identifiers can be very long resulting in the IPU staying on the incorrect path longer, leading to decreased performance. When redirects are infrequent, the impact of using commit fragment identifiers is minimal, as the IPU remains on track most of the time.

We evaluate the usage of commit fragment identifiers in the next chapter in Section 5.11.

4.6 Implementation Aspects

4.6.1 Location/Placement of the IPU

The IPU can be positioned at various points within the processing pipeline. Once the FT has been populated, the IPU is only loosely coupled to the processor, requiring fragment IDs. This communication becomes less frequent, as quantified in the next chapter. Given its relatively decoupled operation, the IPU can be placed either alongside the processor/L1i or near where all active code resides, such as the Last Level Cache (LLC).

Positioning the IPU alongside the processor/L1i would allow it to reuse the paths used for installing missing blocks in the L1i, similar to a typical prefetcher, thereby pulling blocks of code into the L1i. Operating alongside the LLC, on the other hand, would involve accessing blocks of code from the LLC and pushing them to the L1i. Presending is most effective when block movement is frequent between the LLC and L1i. Given that the LLC is already heavily involved in the movement of instruction blocks for these applications, it might be beneficial to consider placing the IPU close to the LLC.

4.6.2 Fragment Table Set Associativity

The Fragment Table (FT) is a set-associative table. When a new fragment is installed in the FT, the table is accessed set-associatively to find an appropriate entry for the newly created fragment. This associativity helps minimize conflicts among the various fragments present in the table.

The IPU frequently accesses the FT to obtain information about a fragment, such as the blocks it accesses and the identifier of the successor fragment. A reasonably sized FT can capture the information for most active fragments in the program, meaning the placement of a fragment within the FT is unlikely to change during the operation of the IPU. The IPU has the option of accessing the successor fragment set-associatively on all accesses. However, these set-associative accesses are likely to consume more energy and increase access time.

Alternatively, given that the placement of FT entries is unlikely to change after their creation, the successor fragment identifier could encode where to obtain the next fragment information from, including way information.Encoding the fragment identifier as a pointer to another entry in the FT helps improve access time and reduces the energy consumption of most FT accesses. Set-associative accesses can be made during the creation of FT entries. They can also be made during IPU redirects, where the IPU must restart sequencing from the fragment identifier obtained from the PFFQ rather than from the FT.

4.7 Summary

This chapter provided a comprehensive description of the Instruction Presending scheme. We began by detailing the various components of the program representation utilized by the scheme. Next, we discussed the hardware data structures required to capture this representation. We then explained how this representation is used to proactively move blocks of code from a lower-level cache to the L1i. Finally, we described the construction of these tables and concluded with specific implementation aspects.

Chapter 5

Presending Evaluation

In this chapter, we present an evaluation of the Instruction Presending (abbreviated *Send*) scheme. We begin by describing the simulation setup in Section 5.1. Next, we quantify the storage required to maintain the shadow program representation used by the Send scheme in Section 5.2. Following this, we provide an outline of our evaluation approach. Finally, we present a detailed evaluation of the Send scheme.

5.1 Simulation Setup

Below, we describe the simulator and benchmarks used, the baseline microarchitecture simulated, and the configurations of two prefetching schemes against which we compare our approach.

To evaluate the Presending technique, we use the ChampSim simulator and all 100 server benchmark traces (as mentioned in Chapter 2). We further group the benchmarks into five categories based on their L1i MPKIs (Misses Per Kilo Instructions) with a baseline instruction cache size of 32KB. Bins I, II, III, IV, and V correspond to L1i MPKIs of 0-15, 15-30, 30-45, 45-60, and >60, containing 16, 45, 6, 23, and 10 benchmarks, respectively.

Processor Decoupled Front-end	
Width	6 instructions
Fetch queue	192 instructions
Decode queue	60 entry
Dispatch queue	60 entry
Branch target buffer (BTB)	8K entries
Target cache	4K entries
Return Address stack	64 entries
Branch penalty	2 cycles (decode stage)
Branch Predictor	Hashed Perceptron 16 Tables, Each Table Size - 4KB Maximum History Length - 232 Minimum History Length - 3
Processor Back-end	
Execute width	4 instructions
Retire width	5 instructions
Re-order buffer	352 entries
Load, store queue	128, 72 entries
Memory hierarchy	
L1i	32KB, 8-way, 4-hit cycles, 16 MSHR
iTLB	64-entry, 4-way, 1 hit cycle
L1-D cache	48KB, 12-way, 5 hit cycles, next-line;
L2 cache	512 KB, 8-way, 10 hit cycles, spp-dev
L2 BTB	16K-entry, 8-way, 8 hit cycle
L2 TLB	2K-entry, 8-way, 8-hit cycle
L3 cache	2MB, 16-way, 20 hit cycles, no pref
DRAM	4 GB, one 8B channel, 1600 MT/s

Table 5.1: Baseline Microarchitecture Parameters

The simulated configuration, shown in Table 5.1, implements a decoupled front-end, modeling FDIP, with a BTB (Branch Target Buffer) and a Target Cache used to predict the targets of indirect branches [23], and a RAS (Return Address Stack). The simulated configuration employs a single-level, 1-cycle BTB, which is more aggressive than designs that use a small L1 BTB corrected by a larger BTB with higher access latency. Cache blocks are sent from L3 to L1i (we describe a rationale for the same in Section 5.5). All schemes are implemented on top of FDIP.

We compare our scheme against two highly effective prefetchers, namely Shotgun (SG) and the Entangling Prefetcher (EP) [45, 64]. For Shotgun, we use a configuration with a 4K U-BTB and a 1K-entry C-BTB, amounting to a total storage of approximately 64KB. We evaluated smaller configurations of SG but found them to be less effective. Additionally, we simulate a slightly more aggressive version of SG compared to what was described in [45]. For Send, when we have discontiguous regions for a fragment that would reside in the ORT (Overflow Regions Table), these discontiguous regions continue to be accessed in a single step (as part of the fragment). In the implementation for SG, they would reside in multiple U-BTB entries and be accessed with multiple accesses to the U-BTB. In our implementation, we process multiple entries of the U-BTB in a single step, similar to Send. For the second prefetching scheme (EP), we use the configuration with a storage overhead of 77.44KB [64].

5.2 IPU Structures and Operation

We describe the sizes of the various entries in different structures used by the IPU (FT, DTT, and ORT), as shown in Table 5.2. The data entry within the FT comprises 2B for the block address (L3 block frame), 0.5B for the block count (up to 16 contiguous blocks in the memory address space), 1B for the instruction count, and 1.5B for the next fragment (a total of 4.5B per frag-

Structure	Entries	Entry Size	Storage
FT	4096	12.375B	50688B
DTT	256	3.375B	864B
ORT-2	1024	6.375B	6528B
ORT-4	256	11.375B	2912B
ORT-16	128	41.375B	5296B

Table 5.2: Table Storage Requirements

ment), for both the call and return fragments. The tag entry consists of 1.75B for the tag and a valid bit.

The data entry in the DTT consists of 1.5B for the next fragment and 2 aging bits for both paths. The tag entry in the DTT comprises 1.375B for the tag and a valid bit, resulting in a storage requirement of approximately 3.375B per DTT entry.

We use three ORT tables: ORT-2, ORT-4, and ORT-16, which can hold up to 2, 4, and 16 code regions, respectively. Each code region comprises 2B for the block address (L3 block frame) and 0.5B for the block count, similar to the code region within the FT. ORT-2 consists of two code regions, which take up 5B, and a tag and valid bit that take up 1.375B, bringing the total size of the entry to 6.375B. ORT-4 consists of four code regions, which take up 10B, and a tag and valid bit that take up 1.375B, bringing the total size of the entry to 11.375B. ORT-16 consists of sixteen code regions, which take up 40B, and a tag and valid bit that take up 1.375B, bringing the total size of the entry to 41.375B.

All tables (FT, ORT, and DTT) are 8-way set associative. While associativity helps minimize conflicts during the placement of fragments, accesses are predominantly direct (given that redirects are infrequent, as quantified in Section 5.4). Including the storage for IPUS and the queues used, the total storage amounts to approximately 68KB with 4K FT entries.

Table 5.3 shows the total storage with different FT entries. With 3K FT entries,

Entries	Storage
2048	43.2KB
3072	55.6KB
4096	68KB

Table 5.3: Table Storage Requirements Varying FT size

the total storage amounts to approximately 55.6KB, and with 2K FT entries, the total storage amounts to approximately 43.2KB. Clearly, FT size is key to bringing down storage, if important, so we evaluate the impact of FT size on the frequency of redirects, a key metric for Presending in Section 5.4.

When an FT entry has a fragment with a separate region, the OF bit is set. Additionally, the block address is replaced with a pointer to the ORT entry to facilitate direct access to the ORT. It is important to note that only a single ORT entry is used per FT entry. This means that for fragments accessing separate regions in addition to the adjoint region and having their Overflow (OF) bit set, only a single ORT table (either ORT-2, ORT-4, or ORT-16) is used to encode all the separate regions accessed. The ORT tables are not cascaded to simplify access, allowing for direct lookup to the ORT entry (instead of a set associative lookup) from the respective ORT using the pointer in the FT entry, accessing all code regions from a single entry. While this approach simplifies ORT access, it could be expanded to allow cascading; however, this dissertation does not explore this alternative.

To tolerate an L1i miss latency of 20 cycles (which corresponds to the L3 access latency) with a 6-wide fetch, Send keeps 20×6=120 instructions ahead. This is the default configuration that we evaluate. We do study the variation of this parameter in Section 5.6.2.

The default configuration for Send explores a second path when it encounters a fragment with multiple successor fragments during high-level sequencing, limiting the total number of active tracks to two at any given time. We explore alternatives and their implications in Sections 5.4.1 and 5.5.1.

5.3 Results Outline

Key components of the evaluation include:

- **High-Level Sequencing Accuracy**: Since a fundamental aspect of the design is high-level sequencing, we begin by providing a detailed study of high-level sequencing accuracy, quantified by the frequency of redirects.
- **Block Movement and Overheads**: High-level sequencing is primarily used to move blocks of instructions to the L1i. Therefore, we next evaluate block movement and the associated overheads in detail, as well as the impact of different parts of the design on block movement. Furthermore, we evaluate two key components of Send: the Block Temperature Table (BTT), which is used to control the movement of blocks from the secondary cache to the L1i, and the Pseudo Inclusion Bit Table (PIT), which is consulted to check for the presence of a code block in the L1i.
- **Timeliness of Instruction Delivery/Cycles Waiting for an Instruction**: Timely delivery of instruction blocks is crucial. Thus, we evaluate the timeliness of the scheme, quantified as cycles that the processor is waiting for an instruction. We also evaluate the ability to stay ahead when the latency of L3 cache is increased. We also evaluate this with an aggressive backend.
- **IPC Performance Benefits**: We assess the IPC (Instructions Per Cycle) performance benefits of the scheme. We also assess the IPC benefits with an aggressive backend.
- **Creation and Updates of FTs**: We study the percentage of dynamic fragments processed that create or update the FT to give an idea how much communication with the processor is needed to create and maintain an FT.
- **Small Cache Sizes**: We study the miss reductions for Send with smaller cache sizes. We also study the cycles the processor is waiting for an

instruction for Send with smaller cache sizes.

– **Fetch Fragment IDs and Wrong-Path Execution**:
  We study some properties of dynamic fragment-level control flow to understand how redirects are likely to be affected in the presence of wrong-path execution.

– **Usage of Retired Fragment IDs**: We study the effect of using retired (commit) fragment IDs to stay sychronized with the processor in place of Fetch Fragment IDs.

5.4 Frequency of Redirects

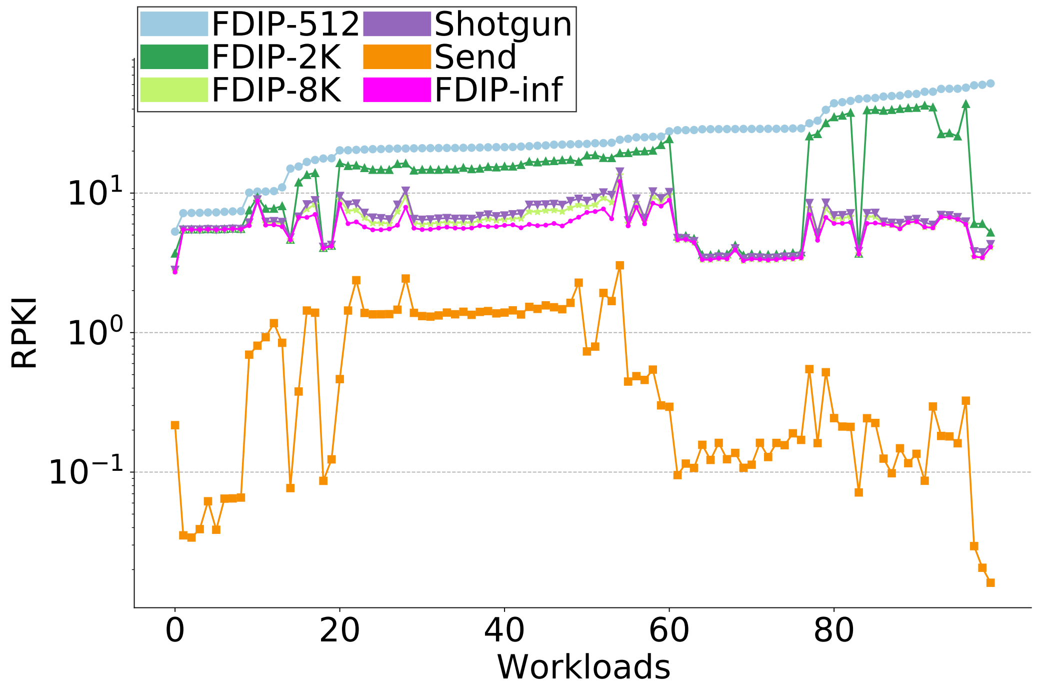


Figure 5.1: RPKI FDIP and Send

Fundamental to the performance of the Send scheme is its ability to run ahead in the instruction stream accurately at a high level (fragment level), ensuring the timely delivery of cache blocks containing the needed instructions. For this purpose, our focus is on determining the blocks of instructions the processor is likely to reference, rather than the precise instruction stream. Redirects refer to inaccuracies in sequencing. For Send, this happens when the processor fetches a fragment which was not sequenced by Send. More specifically, when the fragment inserted by the processor in the PFFQ was not found in the UFQ. A high accuracy in sequencing high-level control flow results in very few redirects.

Figure 5.1 quantifies the Redirects Per Kilo Instructions (RPKI) for six schemes. The Y-axis represents the RPKI (in log scale), and the X-axis represents the 100 benchmarks studied. It quantifies the RPKI for FDIP with BTB sizes of 512 entries, 2K entries, 8K entries, and an infinite-sized BTB (FDIP-512, FDIP-2K, FDIP-8K, and FDIP-Inf). It also quantifies the RPKI for Shotgun (SG) and Send. FDIP relies on using a large BTB and an accurate direction predictor to reduce the number of redirects in the front end, enabling the correct resolution of upcoming control flow. Increasing the BTB size significantly reduces the RPKI for many of the benchmarks, especially when increasing from 512 to 8K (a result observed previously) entries. However, even with an infinite BTB, the RPKI remains in the single digits due to redirects resulting from branch direction mispredicts.

SG reorganizes the BTB to reduce BTB misses, but it still relies on the PC determination logic to correctly go past all control instructions (using a branch predictor and the BTB). The RPKI for SG is similar to that of FDIP-8K.

Send achieves an RPKI that is an order of magnitude lower than even FDIP with an infinite BTB. For example, for server039, the RPKI is 61 for FDIP-512. It falls to 5.22 for FDIP-2K, 4.1 for FDIP-8K and FDIP-Inf, and is 4.3 for SG. With Send, it is 0.02. Similarly, for server023, the RPKI is 44 for FDIP-512, 35.05 for FDIP-2K, 6.5 for FDIP-8K, 6.05 for FDIP-Inf, and is 6.93 for SG. With Send, it is 0.24.

Send maintains very low redirects in many cases because it does not need

Metric	(I)	(II)	(III)	(IV)	(V)
FDIP-Inf	5.50	6.25	5.23	4.29	5.12
SG	5.63	7.69	5.81	4.56	5.40
Send	0.15	1.08	0.19	0.15	0.09

Table 5.4: Redirects Per KI

Metric	(I)	(II)	(III)	(IV)	(V)
Control Instructions/fragment	5.4	5.2	4.6	4.2	3.6
Control Instructions in Adjoint Region/fragment	3.8	4.0	3.8	3.5	3.1

Table 5.5: Control Instructions in a Fragment

to precisely determine the outcome of every control instruction, unlike the other schemes, as it sequences fragment-level control flow. Most fragments have a unique successor fragment, and only a few fragments have more than one successor, in which case Send follows a second path. Redirects occur only when Send is unable to record a successor for fragments with more than two successors, or because it limits the paths explored to two. We also present the Redirects for Send to highlight another key result.

Table 5.4 presents the average RPKI for the five different MPKI bins, for FDIP-Inf, SG, and Send. We observe that RPKI are in the single digits for both FDIP-Inf and SG. SG is able to bring the RPKI close to FDIP-Inf on all bins. Send is able to bring down the RPKI further by an order of magnitude on bins I, III, IV, and V, but exhibits a relatively higher RPKI for bin II. From the table, it is evident that Send is particularly effective at maintaining a low RPKI for applications with a very high L1i MPKI or frequent movement of code between a lower-level cache and the L1i, notably bin IV and V.

Next, we present a characteristic of dynamically executed fragments in Table 5.5, summarized as averages for the five different MPKI bins. Row 1 shows that there are, on average, 4-6 control instructions enclosed within a fragment for the different benchmark bins, as noted in Chapter 3. Row 2 shows that,

Metric	(I)	(II)	(III)	(IV)	(V)
DTT	6.11	9.49	6.69	5.69	4.06

Table 5.6: DTT Accesses

on average, there are 3-4 control instructions enclosed in the adjoint region of a fragment. This indicates that Send, which processes a fragment in each step, goes past this many control instructions in every step, many of which are contained in the adjoint region of the fragment. In contrast, FDIP must accurately go past each control instruction (involving accurate direction predictions and BTB accesses) that Send processes in a single step.

We next conduct further studies to understand redirects for Send specifically.

5.4.1 Effect of Exploring Second Path

Table 5.6 shows the percentage of FT accesses that access the DTT across different MPKI bins. This percentage is relatively small for all bins (4-6.5%) but slightly higher for bin II (9.5%). The remaining dynamic fragments in all bins have a unique successor fragment. For instance, if the percentage is 5%, this means that 95% of FT accesses involve a fragment with a unique successor fragment. When a fragment has a second successor fragment, Send can either proceed down a second path, stop, or continue along one path.

We study the effect on redirects of sequencing high-level control flow differently, as shown in Figure 5.2.

– **Frag-TwoPath:** This configuration allows Send to pursue up to two tracks (paths) at any point of time. This is the default configuration used in all Send simulations. When a fragment has a second successor fragment, Send pursues a second track, as described in Section 4.5.1. The number of outstanding forks is limited to one at any given time.

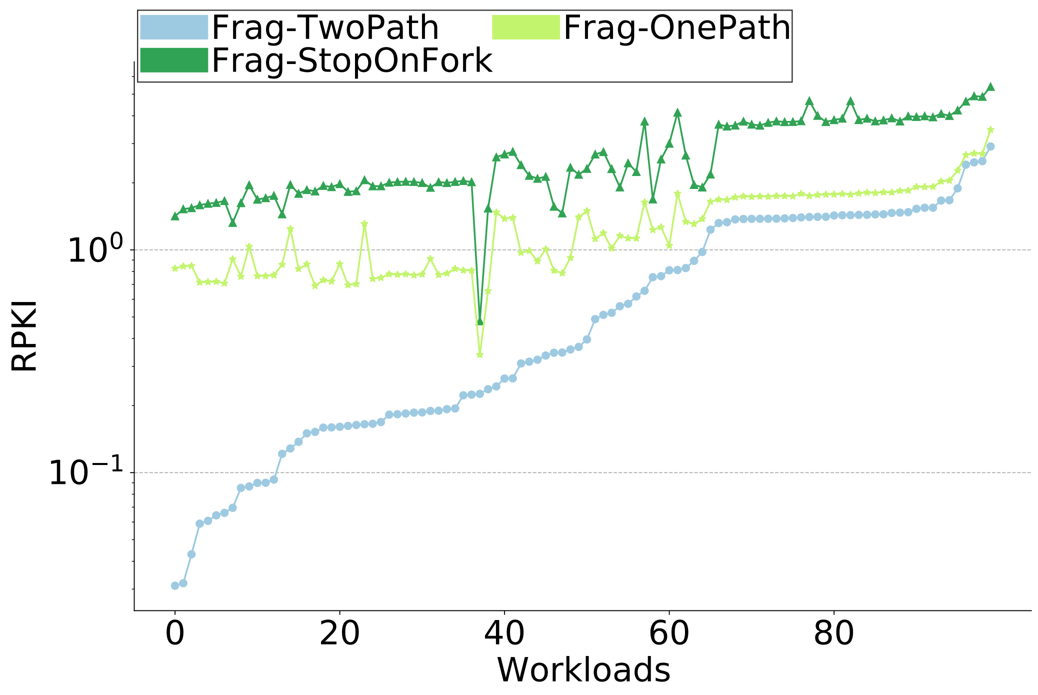


Figure 5.2: RPKI Second Path

– **Frag-StopOnFork:** In this configuration, Send stalls after encountering fragments with more than one successor fragment, until it knows which path the processor goes down.

– **Frag-OnePath:** Here, Send follows one path, which is the previous successor fragment for a fragment with multiple successor fragments.

Clearly, Frag-TwoPath significantly outperforms Frag-StopOnFork by one or two orders of magnitude. This is because Frag-StopOnFork waits for the processor to fetch the successor fragment following the sequencing of a fragment with multiple successor fragments, leading to increased redirects. In this scenario, the fragment fetched by the processor along either path would count as a redirect, as it would not be present in the UFQ. Frag-OnePath narrows this performance gap in some benchmarks by pursuing a single path, thereby reducing the number of redirects experienced after a fragment with multiple successor fragments. This suggests that in an implementation where Send follows only one path, continuing along that path without stopping at forks and using the previous successor fragment to traverse fragment-level control flow is likely to provide substantial benefits in many cases. Nonetheless, pursuing a second path offers a significant reduction in redirects, even compared to Frag-OnePath, in many scenarios. We also quantify the impact on block movement in Section 5.5.1 to demonstrate that Send following a single path provides much of the benefit of following a second path for many benchmarks.

5.4.2 Components of Redirects

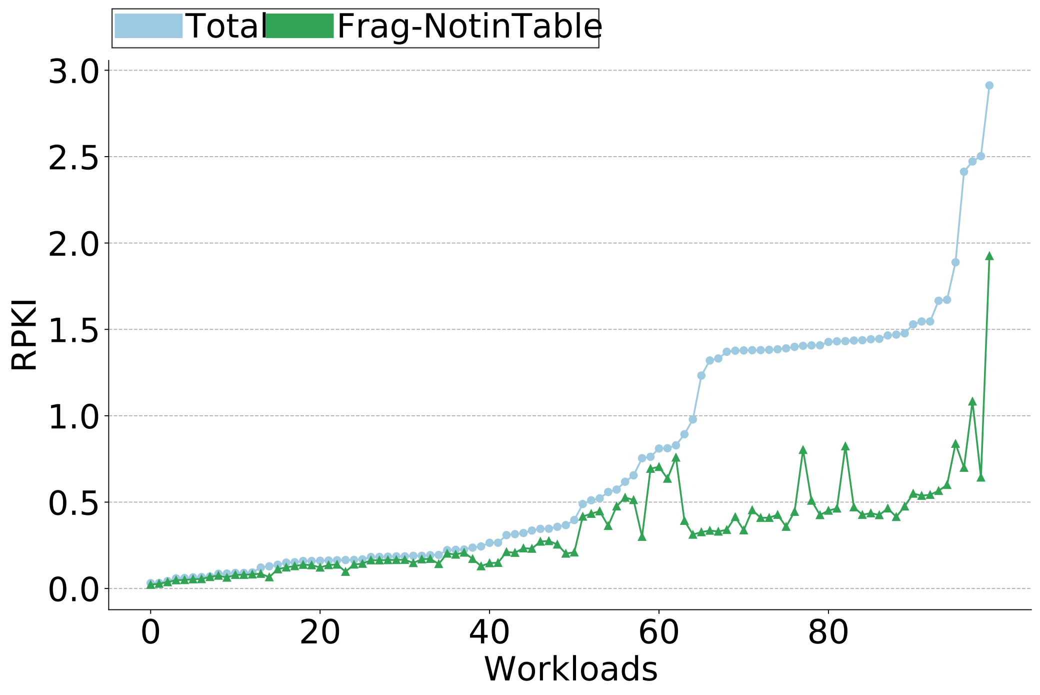


Figure 5.3: RPKI Components

Redirects occur for two reasons:

- We limit the number of successor fragments stored in the table to two, distributed across the FT and DTT.
- We limit the number of paths pursued to two. When there are consecutive fragments with two successor fragments, the scheme goes down the second path and stalls upon encountering the second fragment with two successors.

Figure 5.3 the total Redirects Per Kilo Instructions (RPKI) labelled Total. The figure also contains the RPKI resulting from the inability to store more than two successor fragments labelled Frag-NotinTable. The difference between these two values shows how often redirects occur due to the limitation of pursuing only two paths.

Many applications have low RPKIs below 1, and for most of these applications, the majority of redirects stem from the inability to hold onto the successor fragment in the table.

For applications with higher RPKIs, we observe that the difference between the total redirects and the redirects from the inability to hold onto the successor fragment is quite large. This indicates that these applications experience a non-trivial number of redirects due to the limitation of pursuing up to two paths. While the scheme can be enhanced to overcome this limitation, this enhancement is not explored in this dissertation.

5.4.3 Varying FT Size

Next, we vary the size of the fragment table (which is crucial for capturing fragment-level control) for these benchmarks, as shown in Figure 5.4. This figure examines redirects with 512-entry, 1K-entry, 2K-entry, 4K-entry, and 8K-entry fragment tables.

A 512-entry FT experiences a significantly higher number of redirects (one or two orders of magnitude higher than a 4K-entry FT) due to its inability to capture and reuse fragment-level control flow information. A 1K-entry FT

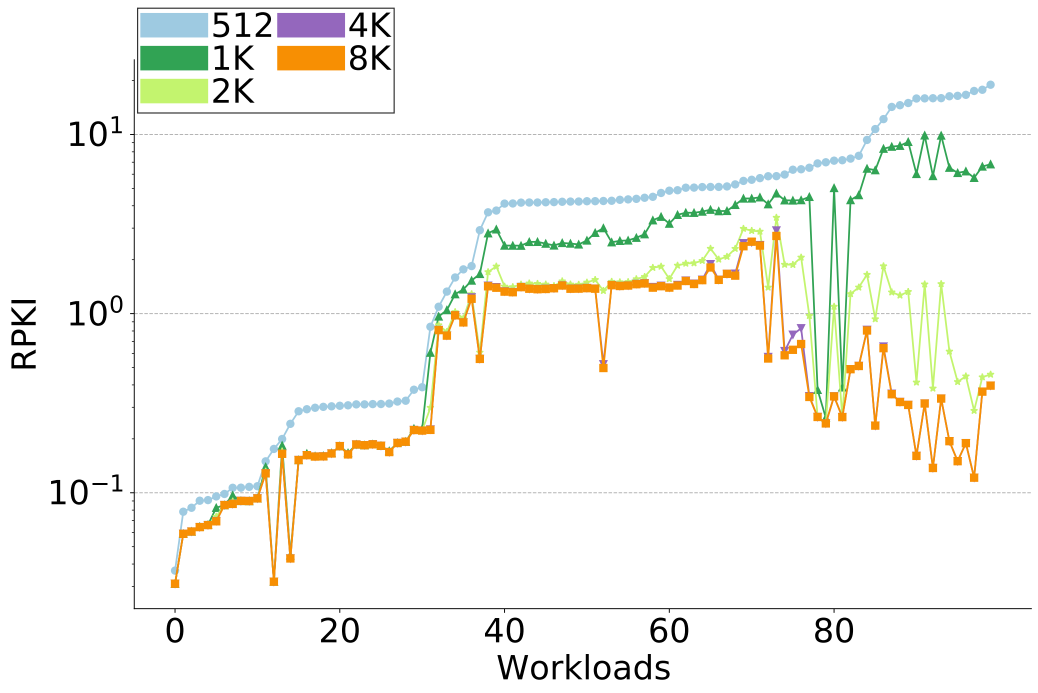


Figure 5.4: RPKI Varying FT Size

shows a reduction in redirects, and a 2K-entry FT performs similarly to the 4K-entry and 8K-entry FTs. The benefits diminish beyond a 4K-entry FT. This result is not surprising, as we observed in Chapter 3 that most applications have 2K-4K active fragments.

5.4.4 Varying FT Associativity

Given that the Fragment Table (FT) is a key data structure holding the information needed to sequence the program at a fragment level, we also study the effect of varying FT associativity for an FT size of 4K entries on redirects, as shown in Figure 5.5. We vary the associativity from 2-way to 16-way. A 2-way associativity results in increased table conflicts, resulting in the eviction of FT entries and, consequently, increased redirects in many cases. A 4-way associativity improves this for many of the benchmarks. There is fur-

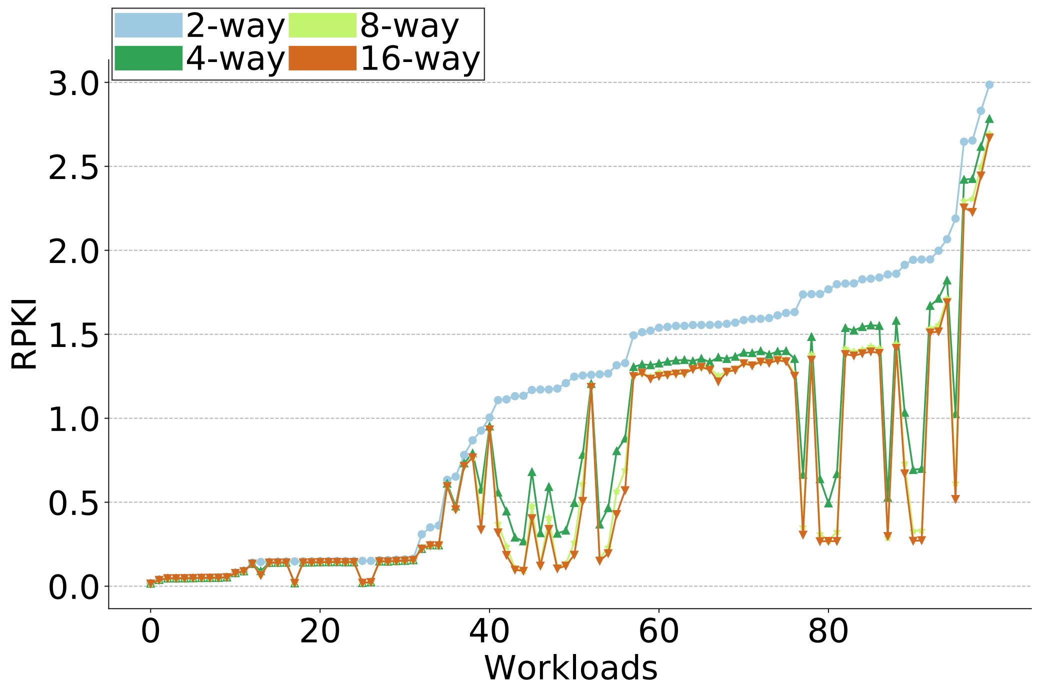


Figure 5.5: RPKI Varying FT Associativity

ther improvement when increasing from 4-way to 8-way associativity, but the benefits diminish beyond 8-way associativity.

Improved hashing schemes to index the FT are likely to help, but we do not explore these alternatives in this dissertation.

5.4.5 Varying Rate of Aging Second Path

The DTT maintains aging bits that aid the IPU in avoiding less frequent (potentially cold) paths. Figure 5.6 studies the effect on redirects of varying the rate at which a path is aged out, where the second path is aged out if the processor does not follow it 1 in 2, 4, 8, or 16 times. Clearly, aging out the path very frequently, such as 1 in 2 times, results in greater redirects. In most cases, aging out the paths 1 in 4 times provides benefits, and 1 in 8 times

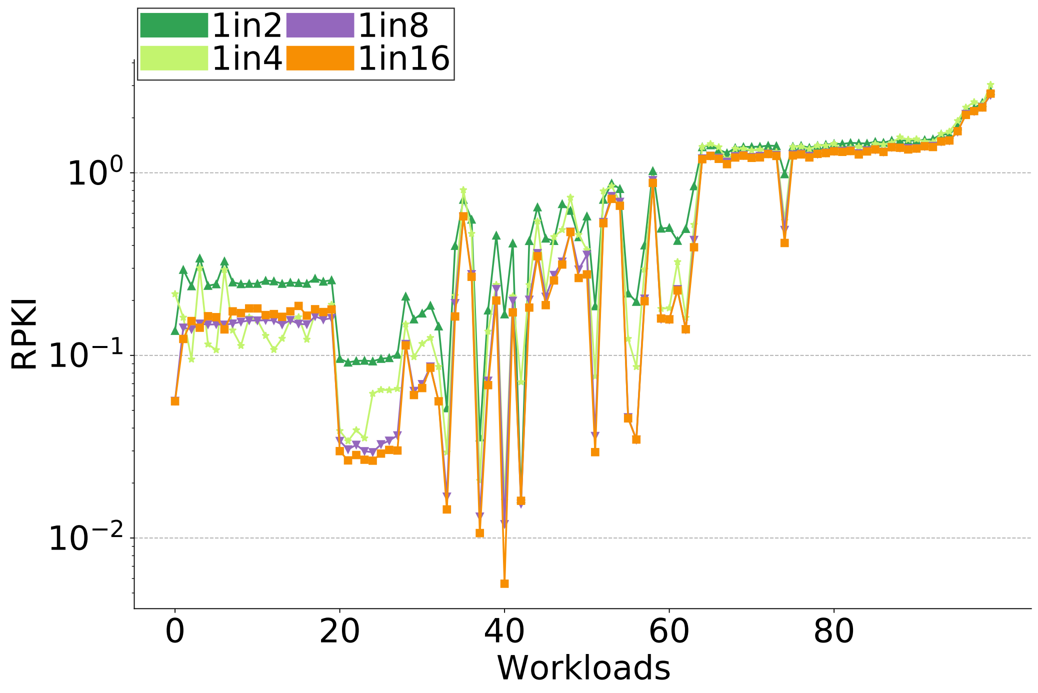


Figure 5.6: RPKI Varying Path Aging Frequency

provides further benefits by continuing to use active paths maintained in the DTT. The reduction in redirects beyond 1 in 8 times is marginal.

We also observe that there are a few workloads where aging out a path 1 in 4 times performs better than 1 in 8 or 16 times. For these benchmarks, the paths remain in the DTT longer, resulting in instances where the IPU encounters back-to-back fragments with two successor fragments and stops. In these cases, aging out a path sooner allows the IPU to continue along one of the two paths that remain active and have not aged out. This reduces redirects compared to stopping or stalling.

5.5 Block Movement to the L1i

In this section, we evaluate the movement of code cache blocks from the L3 cache to the L1i. We focus on the movement of code cache blocks from the L3 cache to the L1i because all active code cache blocks typically fit within a processor's L3 cache, which is generally 2MB or larger in many commercial processors. For example, AMD Zen4 utilizes a 4MB L3 cache per core [15]. Additionally, moving code from the L3 cache to the L1i frees up capacity in the L2 cache for data.

Key metrics we use are:

- **MPKI (Misses Per Kilo Instruction)** - MPKI is a key performance metric. While miss coverage is a commonly used metric to quantify miss reductions, MPKI more effectively highlights the magnitude of these miss reductions. For instance, a reduction from 100 to 10 MPKI is significantly different from a reduction from 10 to 1 MPKI, even though both represent a 90% miss coverage.
- **L3 accesses per KI** - Block movement measures the number of cache blocks transferred between the lower-level cache and the L1i. Accuracy, which refers to the proportion of blocks moved by a scheme that were actually used by the processor, is a commonly used metric to quantify the extra blocks moved between the lower-level cache and the L1i. However, simply showing raw accuracy can be misleading. For instance, an application that was already moving 90 cache blocks every 1000 instructions might have this movement increased to 100 cache blocks with the scheme, while another application with lower block movement might have its block movement increased from 12 cache blocks every 1000 instructions to 24 blocks. Although the accuracy is quite low for the latter (50% compared to 90% for the former), it might be acceptable given the available bandwidth. Movement of blocks involves an L3 cache access, as the movement of code occurs from the L3 to the L1i. Instead of show-

ing accuracy, we quantify the blocks moved using the metric L3 cache accesses per KI to better highlight the magnitude of the increase in block movement resulting from the scheme. The additional L3 cache accesses are a key metric for block movement traffic. This metric not only offers an alternative way to quantify accuracy but also highlights the magnitude of the increase in block movement.

In this section, we aim to study the reduction in misses achieved by Send. Furthermore, we seek to evaluate the benefits in terms of miss reductions resulting from exploring a second path and the use of overflow-region tables (ORTs). We study the usage of a Pseudo-Inclusion Bit Table (PIT), which is crucial for moving blocks of instructions without unnecessarily probing the L1i tags. Additionally, we analyze the block movement traffic. We also examine the Block Temperature Table (BTT), a component used to control the block movement traffic.

We study two configurations for Send: one that employs a Block Temperature Table (BTT) (SendB) and one that does not (SendA). For SendB, we use a 2K-entry BTT (we also study the effect of using different BTT sizes separately). Unless explicitly mentioned, Send refers to SendA. For most of our evaluation, we do not use the Pseudo Inclusion Bit Table and instead probe tags to check for block presence, similar to other prefetchers. We do, however, present an evaluation in Section 5.5.2, where we study the effect of using an inclusion bit and observe that the misses are quite close to those obtained with tag probing.

Base refers to the baseline microarchitecture in all the results we present. We do not show FDIP explicitly in our results relating to misses because FDIP does not reduce MPKI over a microarchitecture without L1i prefetching. Although FDIP misses in the L1i, it can significantly overlap the latency of these misses by running ahead in the instruction stream. This effectively tolerates much of the miss latency, resulting in reduced cycles that the processor waits for an instruction. Overall cycles spent waiting for an instruction is a key metric to evaluate the supply of blocks of instructions, which in turn leads to a performance benefit. We present detailed results for FDIP and the other schemes evaluating these aspects in Sections 5.6 and 5.7.

5.5.1 Miss Reductions

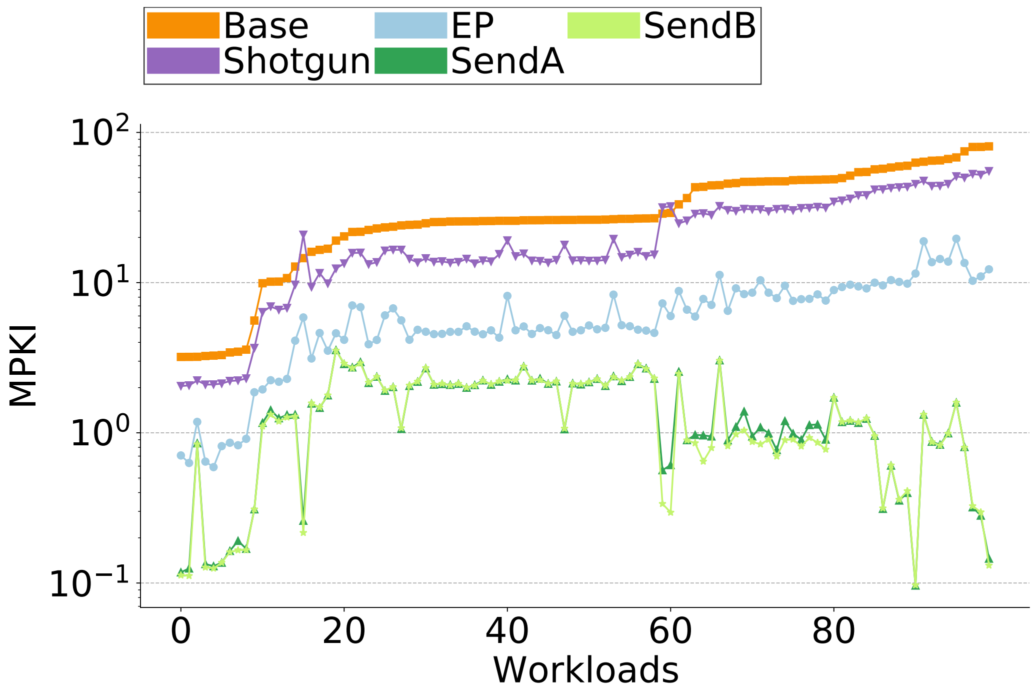


Figure 5.7: L1i MPKI

Figure 5.7 presents the L1i MPKI (log-scale) for various configurations. The configurations include Base (Baseline microarchitecture), Shotgun (SG), SendA, SendB, and Entangling Prefetcher (EP).

In Figure 5.7, we observe that the MPKI for SG is smaller than that for Base. EP achieves a significantly lower MPKI than both Base and SG. Both Send configurations achieve even lower MPKIs and outperform EP by an order of magnitude for some benchmarks, particularly those with high Base MPKIs or those where blocks are frequently moved into the L1i.

For example, for server026, the Base MPKI is 54, while the MPKI for SG is 38. EP reduces this to 9.4, and with Send, it is further reduced to 1.17. Similarly, for server038, the Base MPKI is 80, while the MPKI for SG is 52. EP reduces this to 11, and with Send, it is further reduced to 0.28. The MPKIs for SendA and SendB are very similar in most cases, with either performing slightly better in some instances. SendB primarily provides the benefit of controlling the movement of blocks, which in some cases translates to a small reduction in misses.

Miss Reduction from Exploring Second Path

While we evaluated the effect on redirects from exploring a second path in Section 5.4.1. Here we assess the impact on miss reductions of exploring the second path/track.

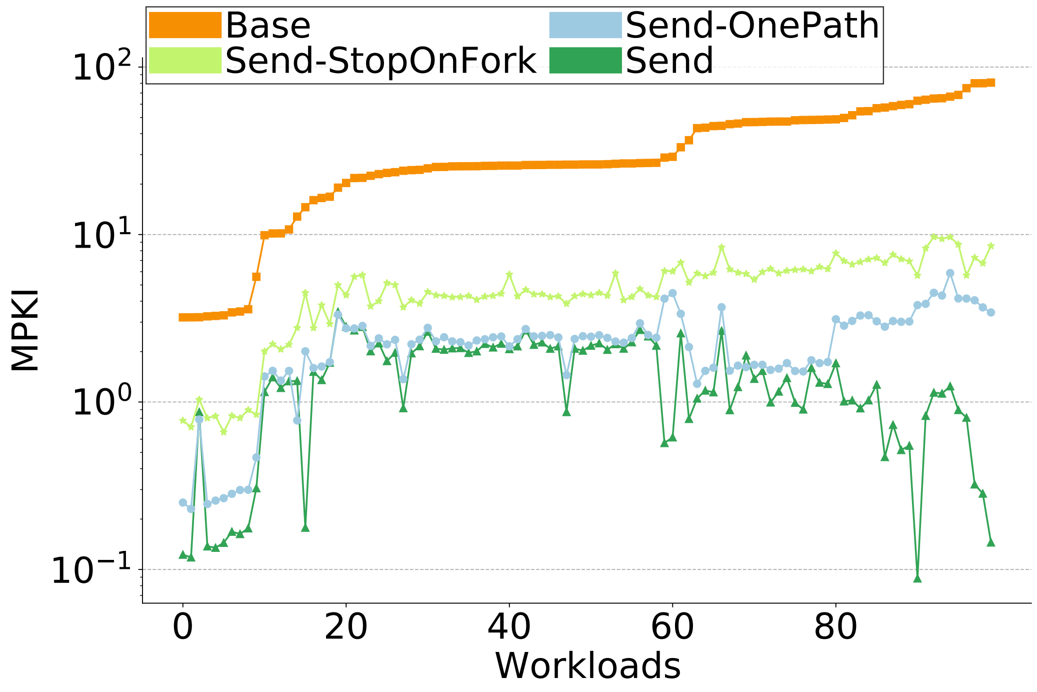


Figure 5.8: Benefits of Pursuing Second Track

Figure 5.8 helps us better understand the benefit Send has from exploring a second track. It plots the MPKI for Base, Send which stalls when it encounters a fragment with multiple successor fragments until the processor reaches the particular fragment to determine the next fragment (Send-StopOnFork), Send which explores one path, choosing the previous successor fragment (Send-OnePath), and the default configuration of Send that explores a second track (Send).

Send-StopOnFork can significantly reduce misses compared to Base. Send-OnePath further reduces misses, bringing performance close to that of Send for some benchmarks. Send achieves additional miss reductions, sometimes relatively higher compared to Send-OnePath, by exploring the second path. Thus, one can use Send-OnePath and still achieve a substantial portion of the miss reductions obtained from pursuing the second path.

Miss Reduction from Using Overflow Region Table (ORT)

Metric	(I)	(II)	(III)	(IV)	(V)
ORT-2	21.21	21.52	13.74	12.68	10.56
ORT-4	2.53	1.63	0.60	0.41	0.11
ORT-16	0.05	0.04	0.07	0.07	0.01

Table 5.7: Dynamic ORT Accesses

Control flow within a fragment often results in accesses to discontiguous blocks, necessitating the use of the Overflow Regions Table (ORT). Rows 1-3 in Table 5.7 studies the percentage of FT accesses that involve the different ORTs for the different MPKI bins. The percentage of accesses to ORT-2 and ORT-4 is relatively higher for bins I and II, which have more control instructions per fragment on average (we had seen in Table 5.5), as we observe in Rows 1 and 2 in Table 5.9. The percentage of accesses to ORT-16 is very small for all the bins.

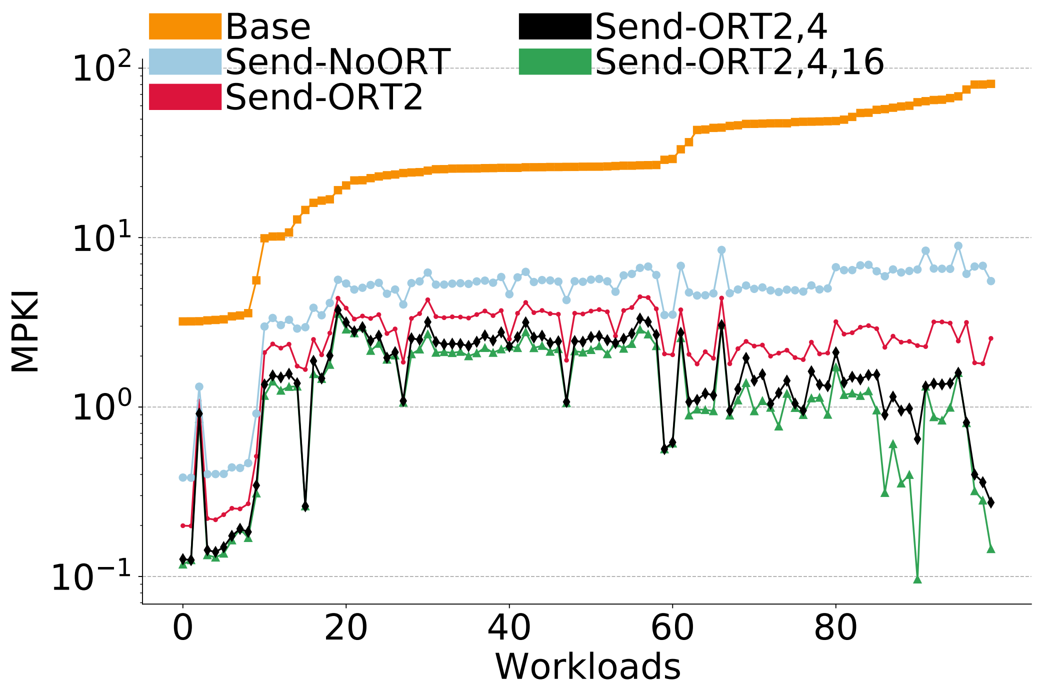


Figure 5.9: Usage of Overflow Regions Table (ORT)

Figure 5.9 analyzes the effect of using ORTs on the L1i MPKI. It presents the MPKI for Base, the MPKI for Send without using ORT using only the adjoint region stored in the FT (Send-NoORT), the MPKI using only ORT-2 that allows for up to 2 code regions per fragment (Send-ORT2), the MPKI using only ORT-2 and ORT-4 that allows for up to 4 code regions per fragment (Send-ORT2,4) and the MPKI using all ORTs (Send-ORT2,4,16). Using only the single region stored in the FT reduces the MPKI to single digits for Send. Utilizing ORT-2 alone further reduces MPKI. The combined use of ORT-2 and ORT-4 brings down misses even more, effectively covering most accesses to separate regions for fragments. The usage of ORT-16 provides a marginal improvement. This is expected, as the ORT-16 is accessed very infrequently.

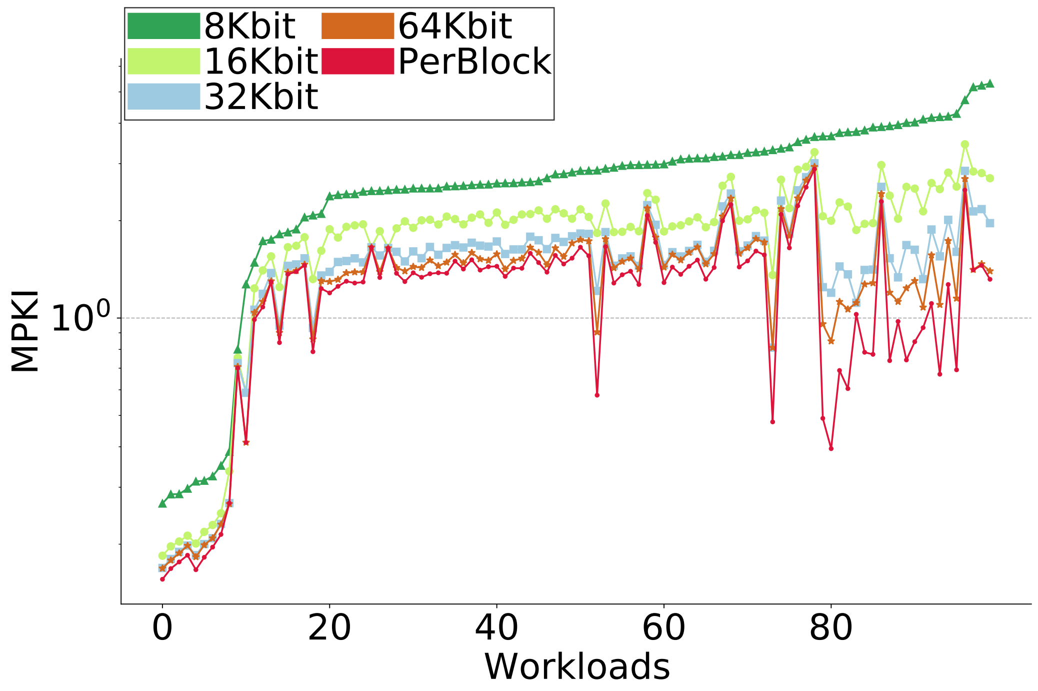


Figure 5.10: MPKI Using Different PIT Sizes

5.5.2 Usage of Pseuso-Inclusion Bit Table

Send can utilize a Pseudo Inclusion Bit Table (PIT), a bit-vector indexed using bits from the block address, separate from the L1i. PIT approximates the presence of blocks in the L1i. The use of a PIT enables decoupled operation, which could also be employed by other prefetching techniques, and can significantly reduce the number of L1i tag probes.

First, we examine the effect of different PIT sizes on L1i MPKI, studied in Figure 5.10. We evaluate PIT sizes ranging from 8Kbits to 64Kbits. Additionally, we assess a configuration where an inclusion bit is maintained along with every block in the L3 cache (PerBlock), amounting to a total of 32Kbits for a 2MB cache with a 64B cache block size. The 8Kbit configuration leaves a significant number of misses uncovered, whereas the 16Kbit and 32Kbit tables provide substantial miss reductions over the 8Kbit table. The 32Kbit configuration approaches the performance of the 64Kbit table on many benchmarks. The per-block configuration provides significant miss reductions on some benchmarks compared to the 64Kbit configuration, primarily due to aliasing in the fixed-size PIT. Per-block configuration benefits from the associativity in the L3 cache, minimizing effects of aliasing.

In our current implementation, the PIT is indexed using the lower n-bits of the block address. Aliasing in the PIT causes missed opportunities for some benchmarks. We leave further enhancements to this to minimize aliasing to future work. Next, we present some results using a 64Kbit-sized (8KB) PIT, to study the impact of using a PIT on L1i tag accesses and on L1i MPKI.

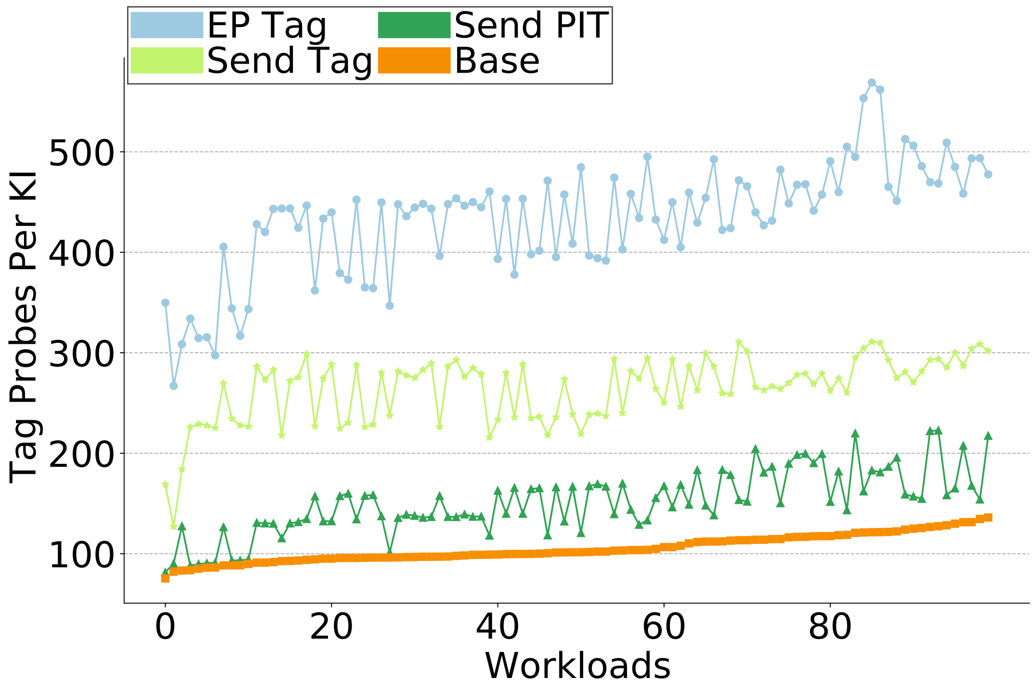


Figure 5.11: Tag Probes Using Inclusion Bit Table

A key benefit of using the PIT is that it reduces the number of L1i tag accesses by utilizing the bit vector to approximate the presence of a cache block in the L1i. We evaluate this aspect next. Figure 5.11 examines the frequency of

L1i tag accesses (Tag Probes) made by Send (with a PIT) as well as configurations that probe the tags (Send Tag, EP Tag) and Base. We quantify the frequency of L1i tag probes as Tag Probes Per KI. EP Tag makes significantly more tag probes compared to Send Tag because EP probes tags on instruction fetch, whereas Send is coupled to a block access. Send PIT can further reduce the tag probes significantly, providing additional benefits and coming close to the tag probes made by Base. Any prefetch scheme will have some extra probes over a baseline because of having to install blocks in the L1i ahead of time.

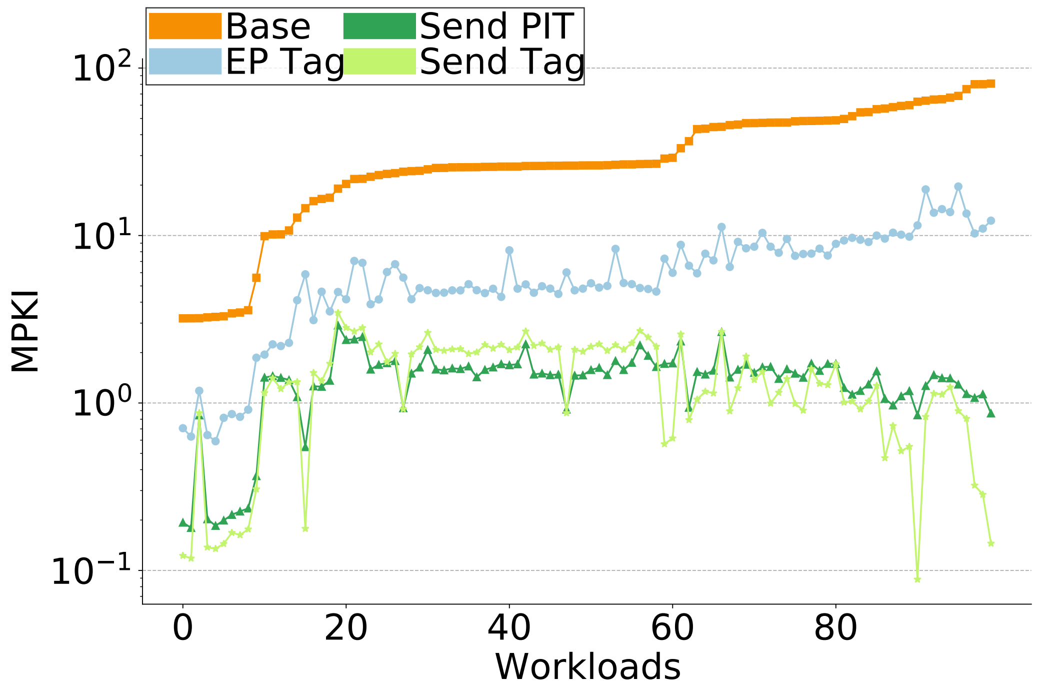


Figure 5.12: MPKI Using Inclusion Bit Table

Figure 5.12 plots the L1i MPKI for the same configurations. Both Send PIT and Send Tag perform signficantly better than EP. From Figure 5.12, we observe that Send Incl Bit performs comparably to Send Tag in many cases. However, in some instances, it experiences more misses compared to Send Tag. This is because Send Tag, during its operation of probing tags to check for presence, also updates the replacement bits for the block. In some cases, this helps prevent the eviction of the block before it is referenced by the processor. On the other hand, Send PIT checks the PIT for presence, and if it indicates the presence of the block, it does not move the block to the L1i. This check happens in advance of when the processor is likely to reference the block, as Send tries to always stay ahead in the instruction stream. In some instances, the replacement policy in the L1i replaces the block from the cache before the block is actually referenced by the processor, resulting in a miss. These are the instances where Send PIT experiences significantly more misses compared to Send Tag.

5.5.3 Block Movement Traffic

To recap, the movement of blocks to the L1i involves the L3 cache. To better quantify the increase in the magnitude of block movements, we use the metric L3 cache accesses per KI. We first present some results to understand the BTT, a data structure that is key to controlling the block movement traffic.

Traffic Control Using BTT

The Block Temperature Table (BTT) is a linear array of 3-bit temperature values used to indicate whether a block is cold or hot. The idea is that cold blocks need not be moved to the L1i. The BTT is indexed using the lower n-bits of the block address. This key data structure can be used to control the block movement traffic between the L3 cache and the L1i.

We first present an evaluation of the traffic with different BTT sizes. Figure 5.13 shows the block movement traffic with no BTT and with different BTT sizes: 1K, 2K, and 4K entries, corresponding to sizes of 384B, 768B, and 1.5KB, respectively, as well as with a block temperature value maintained alongside every code cache block in the L3 cache (PerBlk). Clearly, the use of the BTT can significantly reduce block movement traffic for some benchmarks. For example, in server002, the block movement traffic decreases from 41 to 22.7

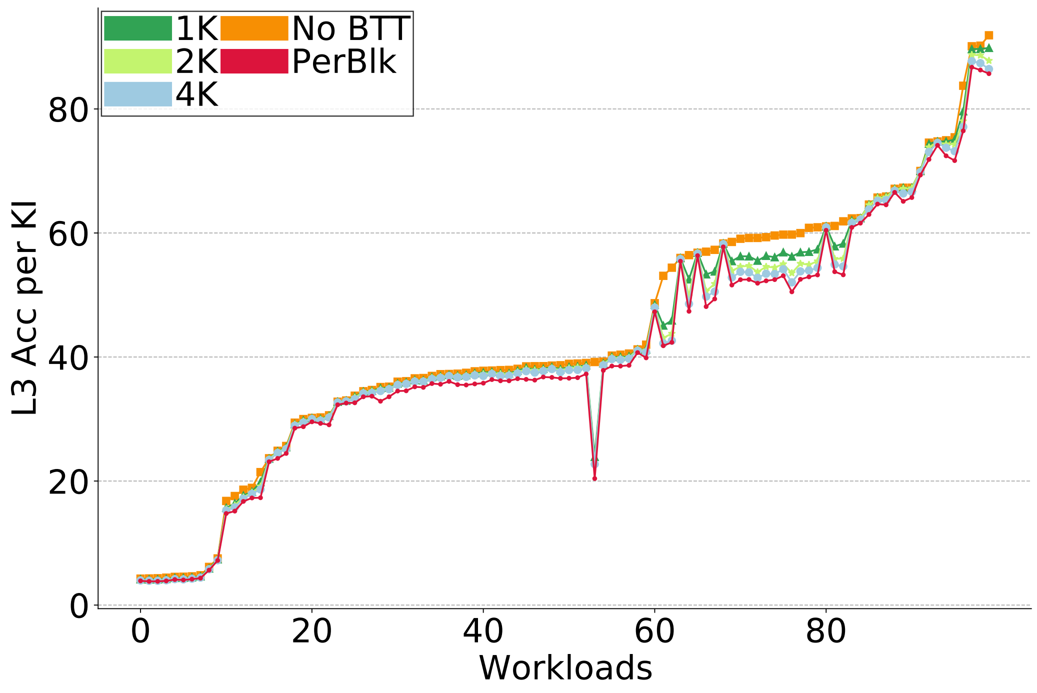


Figure 5.13: Traffic With Different BTT Sizes

L3 cache accesses per KI, with a 2K-entry BTT. The benefit offered by a 1K-entry BTT is quite substantial, and going beyond 1K entries provides only a small benefit. Beyond 2K entries, the benefit is very marginal. For our simulations, we use a 2K-entry BTT.

Traffic Increase With Send

Next, we evaluate the additional block movement introduced by Send as well as other schemes, as shown in Figure 5.14. The Base results are somewhat optimistic because we do not evaluate the effect of wrong-path prefetching given the trace-based infrastructure we are using. The extra traffic for SG is similar to Base, though slightly higher in a few cases. Send exhibits higher traffic in most cases, but the relative increase in traffic is small for benchmarks that already have high traffic. SendB can reduce traffic compared to SendA

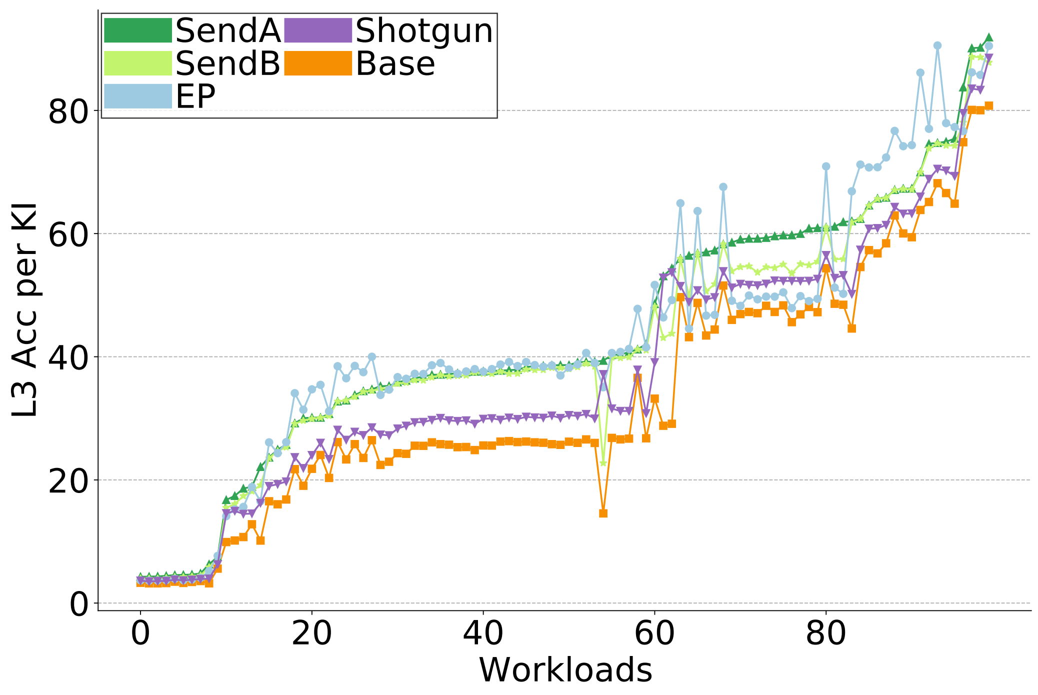


Figure 5.14: L1i Traffic

through the use of temperature bits in some cases, as previously discussed. The extra traffic for EP is also higher compared to Base and SG, and is on par with SendA and SendB.

We next provide a more detailed evaluation of the extra code movement for Send.

Excess Block Movement Breakdown

Blocks can be wastefully sent to the L1i for two reasons:

- The IPU pursues multiple tracks when a fragment has multiple successor fragments.
- A fragment encapsulates all fragment internal control flow, and not all blocks in a fragment may be used based on the outcomes of the control

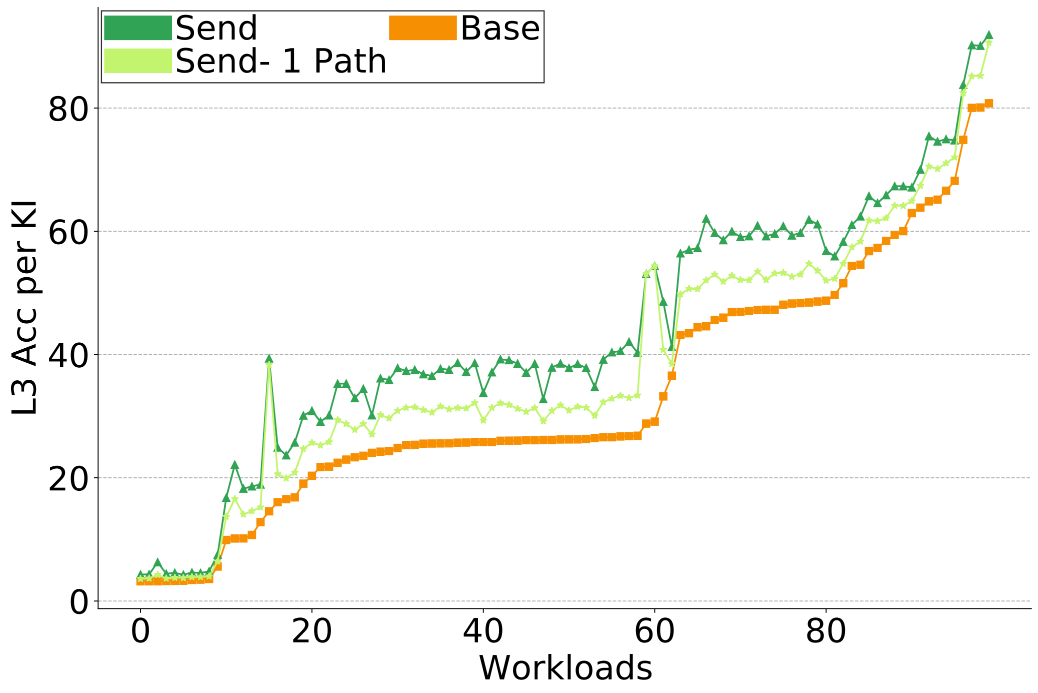


Figure 5.15: Traffic Breakdown

instructions within the fragment.

Figure 5.15 plots the traffic for Base, Send, and traffic from Send along the correct path alone (Send - 1 Path). The difference between Send and Send - 1 Path highlights the extra traffic generated from pursuing a second track/path. The difference between Base and Send -1 Path highlights the extra traffic from encapsulating all fragment internal control flow. For some benchmarks, we observe a significant contribution to extra traffic from pursuing a second path. For some of the benchmarks, most of the contribution appears to stem from encapsulating all the local control flow, as the traffic from Send - 1 Path is almost the same as the traffic for Send.

We also present the traffic for Base, Send - 1 Path, and Send for the different L1i MPKI bins in Table 5.8 to better highlight some trends. The first three data rows present the average traffic for Base, Send - 1 Path, and Send. For

Metric	(I)	(II)	(III)	(IV)	(V)
Base	5.42 (1.00)	24.45 (1.00)	40.65 (1.00)	50.14 (1.00)	69.38 (1.00)
Send- 1 Path	6.98 (1.28)	30.26 (1.24)	46.75 (1.15)	54.99 (1.1)	74.37 (1.07)
Send	8.48 (1.56)	36.05 (1.47)	53.28 (1.31)	60.71 (1.21)	77.70 (1.12)
DTT	6.11	9.49	6.69	5.69	4.06

Table 5.8: Traffic Breakdown for MPKI Bins

each of the fields in these rows, the relative increase in traffic over Base is shown in parentheses. The last data row presents the average percentage of FT accesses that access the DTT. For bins I, II, and III, the relative increase in traffic from Send over Base is significant (over 45% for bins I and II). Additionally, the traffic increase from pursuing a second path is notable (a 23% increase over Base for bin II which is the difference between Send and Send-1 Path) due to more frequent DTT accesses (9.5% for bin II), as these bins have more fragments with multiple successor fragments. The relative increase in traffic for Send over Base in bins IV and V is smaller (12-20%), with similar contributions from pursuing a second path and and from encapsulating all fragment internal control flow. These bins also make fewer DTT accesses.

5.6 Timeliness Evaluation

To recap, timeliness refers to the scheme's ability to fully tolerate the latency associated with fetching instructions from a lower-level cache, thereby minimizing the time spent waiting for instructions. While reducing cache misses is important, the ability to overlap these misses can reduce the average miss latency. Misses can have varying latencies depending on the scheme's effectiveness in overlapping them. The impact on the frontend is primarily determined by how much time it spends waiting for instructions, making the cycles spent by the processor waiting for an instruction a crucial metric. Maintaining a low cycle count in this regard is key to achieving high frontend performance and improving overall application performance. First, we evaluate this aspect, followed by an assessment of the impact of instruction lookahead on timeliness.

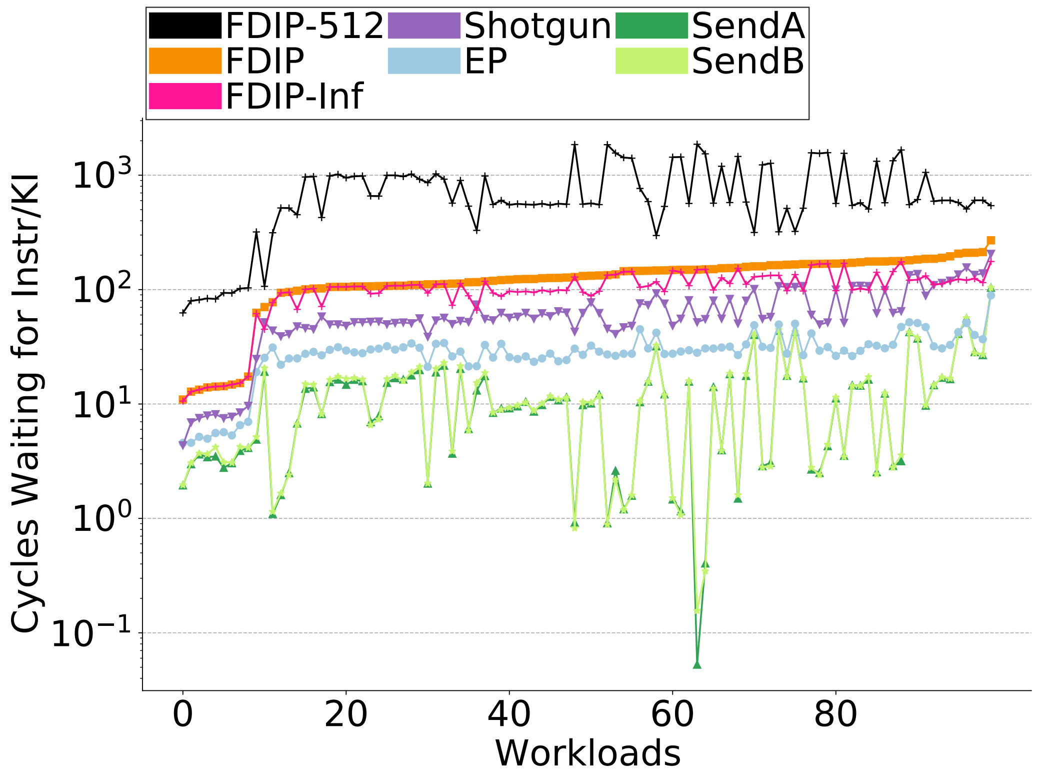


Figure 5.16: Cycles Waiting for Instruction Per KI with 20-cycle L3 Latency

5.6.1 Cycles Spent Waiting for an Instruction

Figure 5.16 quantifies the cycles spent waiting for an instruction to be fetched, expressed as cycles waiting for an instruction per kilo instructions (in log scale). The figure plots these cycles for FDIP-512 (with a 512 BTB), FDIP (with an 8K BTB), FDIP-Inf (FDIP with an infinite BTB), SG, EP, SendA, and SendB. For Send, we use the default configuration with a lookahead of 120 instructions. FDIP-512 experiences an order of magnitude higher cycles waiting for instructions compared to FDIP, as the large BTB (8K-entry BTB) allows

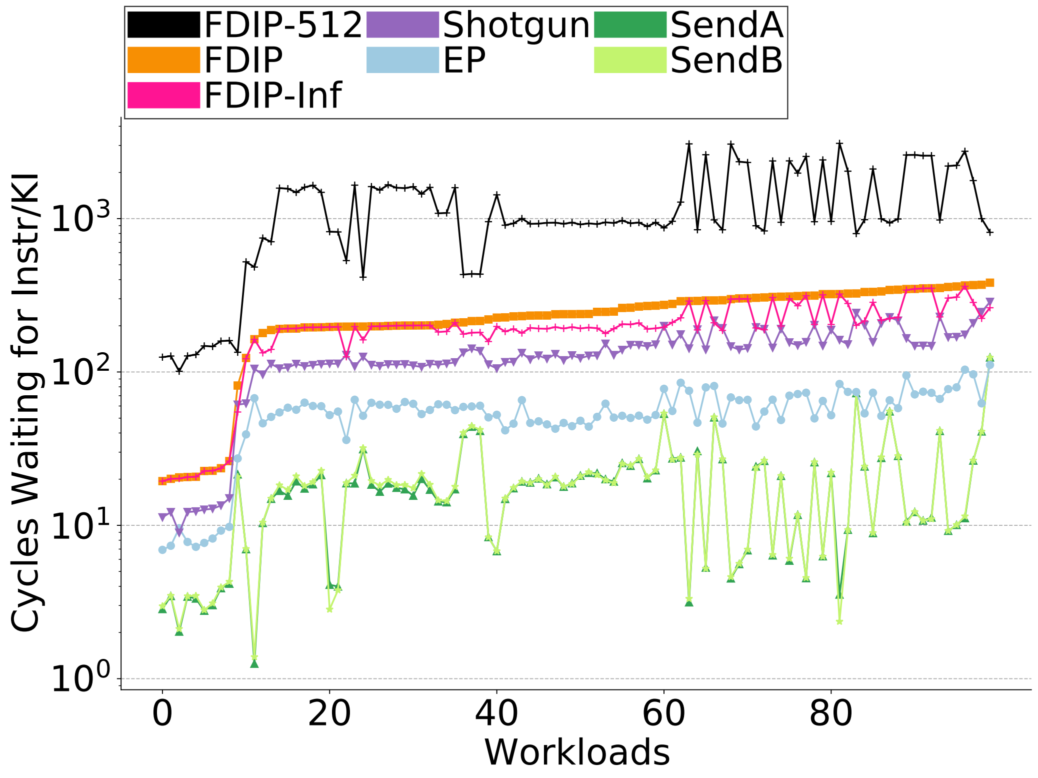


Figure 5.17: Cycles Waiting for Instruction Per KI with 40-cycle L3 Latency

FDIP to effective runahead in the instruction stream, thereby better tolerating the latency of fetching instructions from a lower-level cache. This significantly reduces the waiting cycles. FDIP experiences almost an order of magnitude higher cycles waiting for instructions compared to Send, sometimes even two orders of magnitude higher. FDIP-Inf improves this slightly. SG further reduces the cycles waiting for instructions. EP further reduces the cycles waiting for an instruction. However, both SendA and SendB still provide higher reductions over EP, in many cases an order of magnitude higher compared to EP, often reducing the cycles waiting for instructions to single digits or close to zero. The difference in cycles waiting for an instructions between SendA and SendB is very small.

Similarly, Figure 5.17 quantifies the cycles spent waiting for an instruction to be fetched, expressed as cycles waiting for an instruction per kilo-instructions (in log scale), for the same configurations, with a higher L3 latency of 40 cycles. For Send, the instruction lookahead is increased to 240 instructions to tolerate the higher latency. The trends are mostly similar. Cycles spent waiting for an instruction are higher for the FDIP configurations and SG. We observe that both EP and Send continue to maintain low cycles waiting for instructions (though slightly higher than with an L3 latency of 20 cycles), with Send performing better and maintaining cycles waiting for an instruction in the single digits in many cases.

5.6.2 Sensitivity to Instruction Lookahead

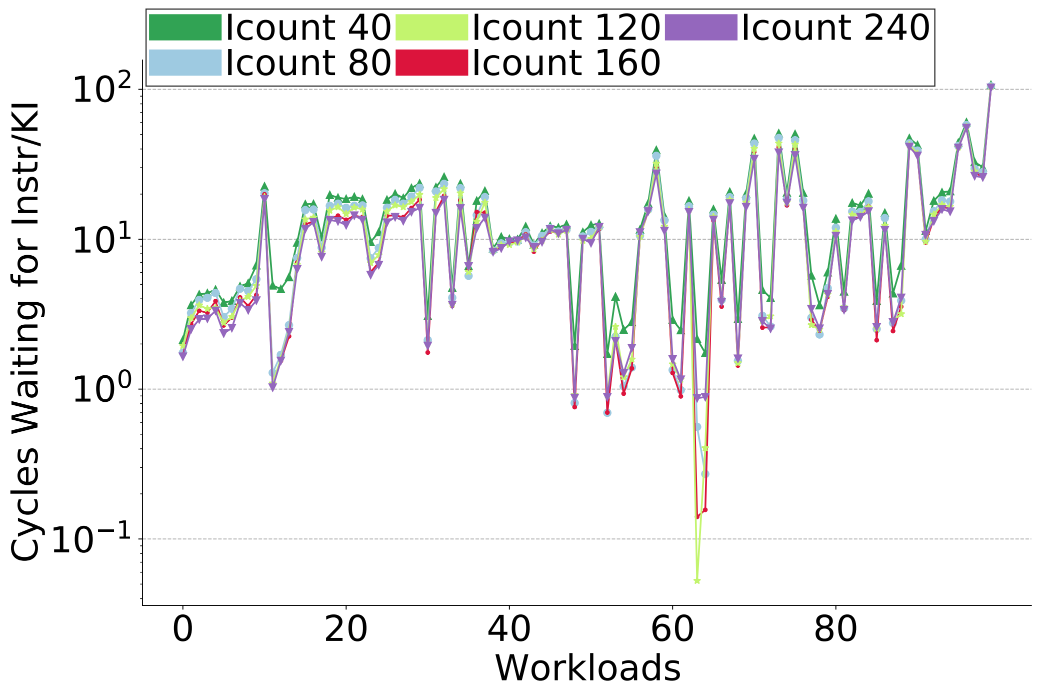


Figure 5.18: Varying Instruction Lookahead with 20-cycle L3 Latency

Instruction based lookahead is a key parameter for Send, determining its ability to tolerate the L3 latency from which blocks are moved to the L1i and

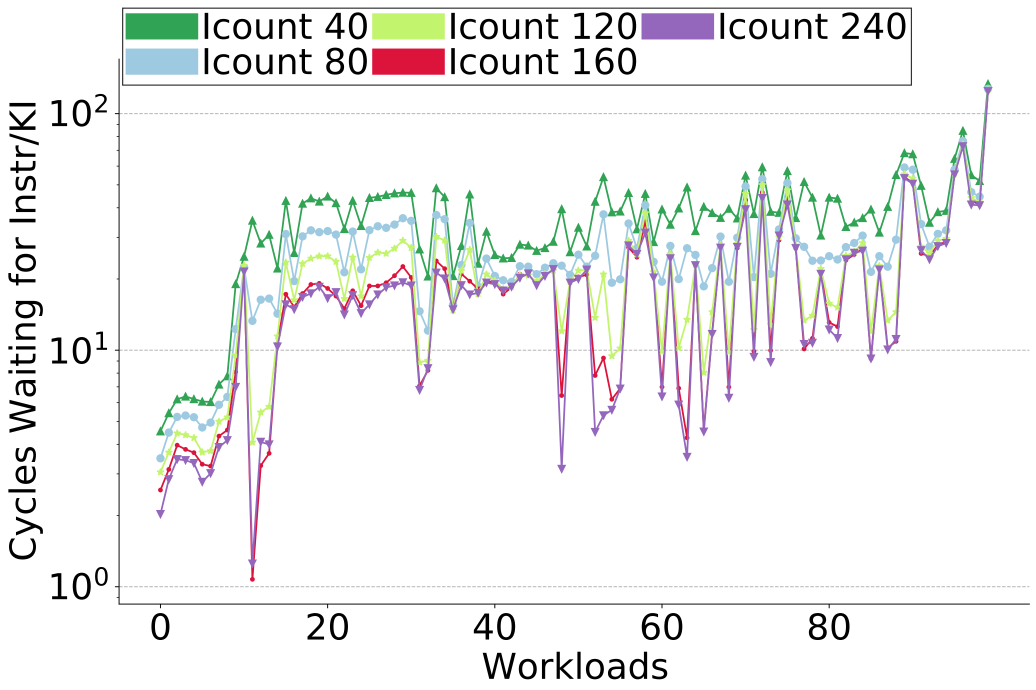


Figure 5.19: Varying Instruction Lookahead with 40-cycle L3 Latency

ensure a timely supply of instruction blocks. For all of our simulations, the default instruction lookahead is 120 instructions. In Figure 5.18, we vary the instruction lookahead for Send from 40 to 240 (Icount40, Icount80, Icount120, Icount160, and Icount240) and show the Cycles Waiting for an Instr/KI in log-scale.

Program phases that fetch 6 instructions per cycle benefit from higher lookaheads. Icount40 brings down the cycles waiting for an instruction to the single digits in many cases. Icount80 improves cycles waiting for instructions further, and beyond Icount120, the cycles waiting for an instruction remains almost the same.

In Figure 5.19, we show the same configurations, this time with an L3 latency of 40 cycles. We expect higher instruction lookaheads to effectively tolerate this increased latency of fetching instructions from a lower-level cache.

Clearly, we observe that there is a significant difference in the cycles waiting for instructions when going from Icount40 to Icount120. Going from Icount120 to Icount160 further provides a significant reduction in many cases. Beyond Icount160, the reduction in cycles waiting is quite small. Send can effectively keep ahead in the instruction stream, and this becomes more important as we try to tolerate higher fetch latencies.

5.6.3 Using Cycle Counts Instead of Instructions

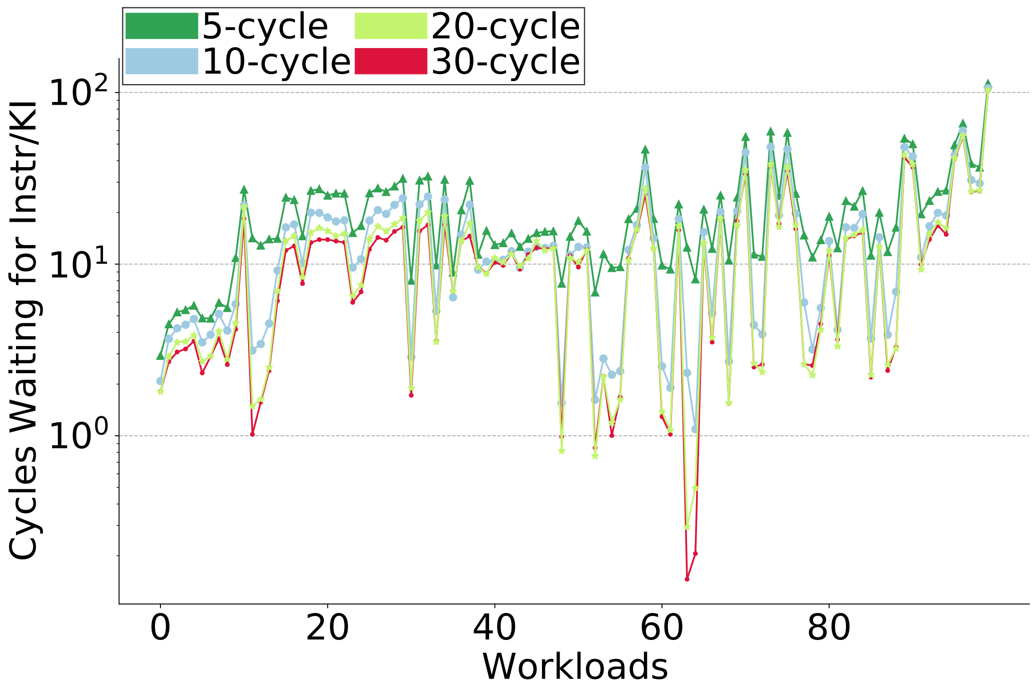


Figure 5.20: Varying Cycle Lookahead with 20-cycle L3 Latency

Next, we study the effect of using fetch cycle counts for a fragment instead of dynamic instruction counts to keep ahead of the processor. This parameter determines the ability to tolerate the L3 latency from which blocks are moved to the L1i. In Figure 5.20, we vary the cycle lookahead for Send from 5 to 30

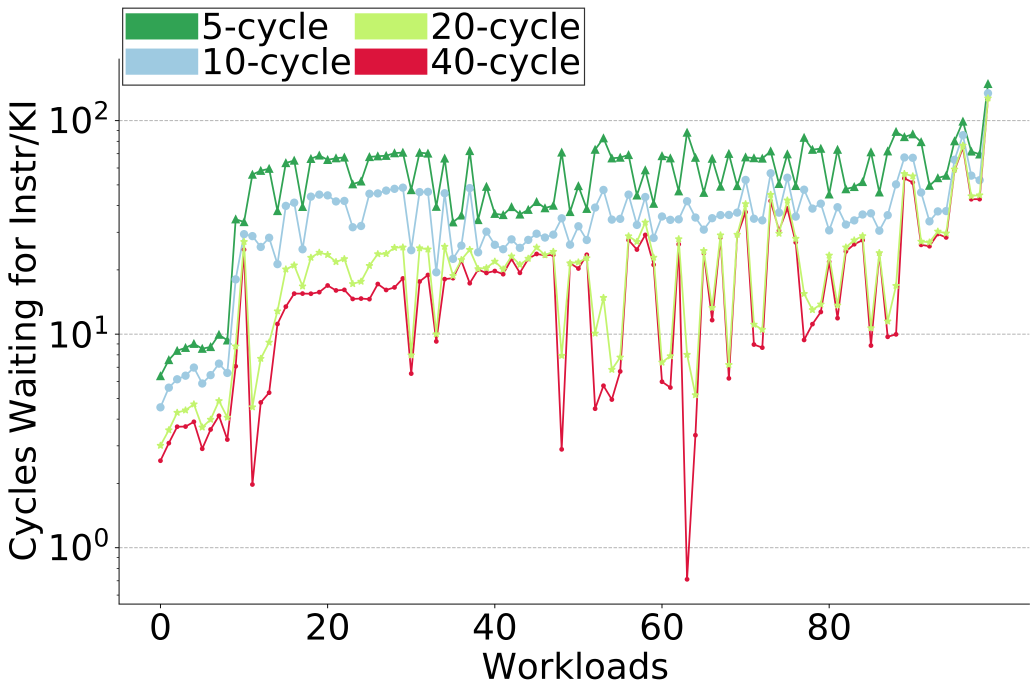


Figure 5.21: Varying Cycle Lookahead with 40-cycle L3 Latency

cycles (5-cycle, 10-cycle, 20-cycle, and 30-cycle) and show the Cycles Waiting for an Instruction per Kilo-Instruction in log scale.

We observe a significant reduction in waiting cycles when increasing the lookahead from 5 cycles to 10 cycles. Furthermore, many benchmarks show additional reductions when increasing from 10 cycles to 20 cycles. However, reductions are marginal beyond 20 cycles. This is expected, as we are tolerating a latency of 20 cycles when fetching instructions from an L3 cache. This illustrates that cycle counts could also be used as an alternative to instruction counts to aid in keeping ahead of the processor.

Similarly, in Figure 5.21, we vary the cycle lookahead for Send from 5 to 40 cycles (5-cycle, 10-cycle, 20-cycle, and 40-cycle) and show the Cycles Waiting for an Instruction per Kilo-Instruction in log scale, with an L3 latency of 40 cycles. A higher cycle lookahead is required to tolerate the L3 latency. Increasing the lookahead from 5 to 20 cycles results in a significant reduction in cycles waiting. Some benchmarks continue to see improvements when increasing the lookahead from 20 to 40 cycles.

5.7 IPC Performance

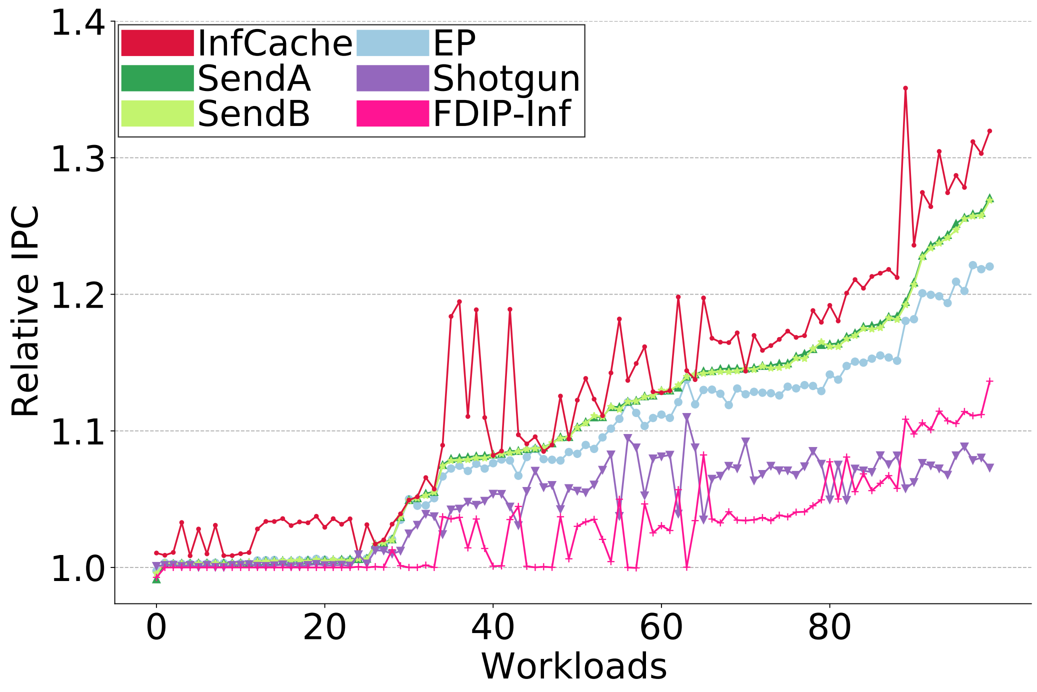


Figure 5.22: L1i IPC

Reducing the cycles spent waiting for an instruction enhances the supply of instructions to the processor's backend, leading to improved IPC performance. Next, we quantify this increase in IPC performance.

The figure 5.22 presents the relative IPC over FDIP with an 8K BTB. The results are shown for Infinite L1i (InfCache), SendA, SendB (with the use of a BTT), Entangling Prefetcher (EP), Shotgun (SG), and FDIP with an infinite BTB (FDIP-Inf).

FDIP-Inf and SG achieve some performance benefits over FDIP, with SG outperforming FDIP-Inf in many cases. As we saw in the previous Section 5.6.1, FDIP-Inf reduces the cycles spent waiting for an instruction slightly compared to FDIP with an 8K-BTB, while SG reduces this even further. One reason for SG's enhanced performance over FDIP is that it is allowed to prefetch blocks spanning multiple UBTB entries, which are part of a fragment in one step, whereas FDIP would prefetch them in multiple steps. EP further reduces the cycles spent waiting for an instruction compared to SG and FDIP-Inf, resulting in a 10-20% performance improvement over FDIP with an 8K BTB in many cases.

Send reduces cycles waiting for an instruction even further in some cases, which we saw in Figure 5.16, leading to a slightly higher IPC performance benefit compared to EP. The cycles waiting differences between SendA and SendB are minimal, so their IPC performance benefits are nearly identical. Send provides an IPC performance benefit of 10-27% over FDIP with an 8K BTB for many benchmarks, approaching the performance of InfCache in many cases. For some benchmarks, we observe greater differences between the performance of Send and InfCache; these are the benchmarks where Send experiences higher redirects (RPKI). We next present data for the different L1i MPKI bins to better illustrate this.

Table 5.9 presents the absolute IPC, RPKI, and Cycles Waiting for an Instruction Per KI averages across different MPKI bins for various configurations. We do not show the RPKI for EP since it is identical to FDIP-8K, as EP does not explicitly sequence the program control flow. RPKI for both SendA and SendB are also identical. The first 7 data rows show the absolute IPC for the different configurations, followed by 4 data rows presenting the RPKI for select configurations. The last 6 data rows show the Cycles Waiting for Instruction Per KI for the various configurations. The InfCache configuration has zero waiting cycles since instructions are always available to the processor.

We observe that the IPC difference between InfCache and Send is minimal

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
IPC					
FDIP-8K	1.32	1.42	0.83	0.77	1.43
FDIP-Inf	1.34	1.51	0.84	0.78	1.43
SG	1.34	1.51	0.86	0.80	1.51
EP	1.36	1.61	0.88	0.81	1.54
SendA	1.36	1.65	0.89	0.81	1.55
SendB	1.36	1.64	0.89	0.81	1.55
InfCache	1.40	1.69	0.90	0.82	1.56
RPKI					
FDIP-8K	5.57	7.11	5.53	4.39	5.13
FDIP-Inf	5.50	6.25	5.23	4.29	5.12
SG	5.63	7.69	5.81	4.56	5.40
Send	0.15	1.08	0.19	0.15	0.09
Cycles Waiting for Instruction Per KI					
FDIP-8K	34.76	145.06	127.00	125.46	154.13
FDIP-Inf	31.96	99.69	110.26	119.29	153.31
SG	19.42	79.28	59.35	51.90	50.82
EP	12.41	30.17	33.30	30.22	31.10
SendA	6.42	12.86	10.99	6.99	1.33
SendB	6.72	12.92	11.39	7.15	1.43

Table 5.9: Absolute IPC, RPKI and Cycles Waiting Per KI for MPKI Bins

for bins III, IV, and V because they maintain very low RPKI, an order of magnitude smaller than FDIP with both an 8K BTB and an infinite BTB, as well as SG. Additionally, the waiting cycles are in the single digits for bins IV and V, slightly higher for bin III. The IPC difference is more significant for bin II due to higher RPKI and a higher cycles waiting for an instruction with Send. The IPC difference becomes more pronounced with an aggressive microarchitecture, as discussed in Section 5.7.1. In bin I, many applications have a low RPKI, but a few have higher RPKI and higher cycles waiting for an instruction, making the average IPC difference between InfCache and Send more noticeable.

Although the performance benefit of Send over EP is small, we shall see in

Processor Decoupled Front-end	
Width	8 instructions
Fetch queue	192 instructions
Decode queue	60 entry
Dispatch queue	60 entry
Branch target buffer	8K entries
Target cache	4K entries
Return Address stack	64 entries
Branch penalty	2 cycles (decode stage)
Branch Predictor	Hashed perceptron
Processor Back-end	
Execute width	10 instructions
Retire width	8 instructions
Re-order buffer	1000 entries
Load, store queue	300, 300 entries
Memory hierarchy	
L1i	32KB, 8-way, 4-hit cycles, 16 MSHR
iTLB	64-entry, 4-way, 1 hit cycle
L1-D cache	Perfect, 3 hit cycles, next-line
L2 cache	512 KB, 8-way, 10 hit cycles, spp-dev
L2 BTB	16K-entry, 8-way, 8 hit cycle
L2 TLB	2K-entry, 8-way, 8-hit cycle
L3 cache	2MB, 16-way, 20 hit cycles, no pref
DRAM	4 GB, one 8B channel, 1600 MT/s

Table 5.10: Aggressive Microarchitecture Parameters (uA2)

Chapter 7 that Send can provide similar performance even with a small-sized primary L1 BTB, while EP relies on a large primary L1 BTB.

5.7.1 Using an Aggressive Microarchitecture (uA2)

Send achieves a low MPKI and a low cycles waiting for an instruction and is not dependent on any processor microarchitectural events, branch direction predictor, or BTB. Therefore, we expect Send to keep up as the demands on the front end grow. To illustrate this, we evaluate Send with an aggressive backend, which puts more pressure on the processor frontend, making the timely supply of instructions even more critical.

For this evaluation, we simulate a machine with the parameters described in Table 5.10. The key changes include the use of an ideal data cache (3-cycle hit latency) that always hits, 8-wide fetch, 10-wide issue, and a 1000-entry reorder buffer. Additionally, we allow multiple-taken branches to be predicted in a single cycle.

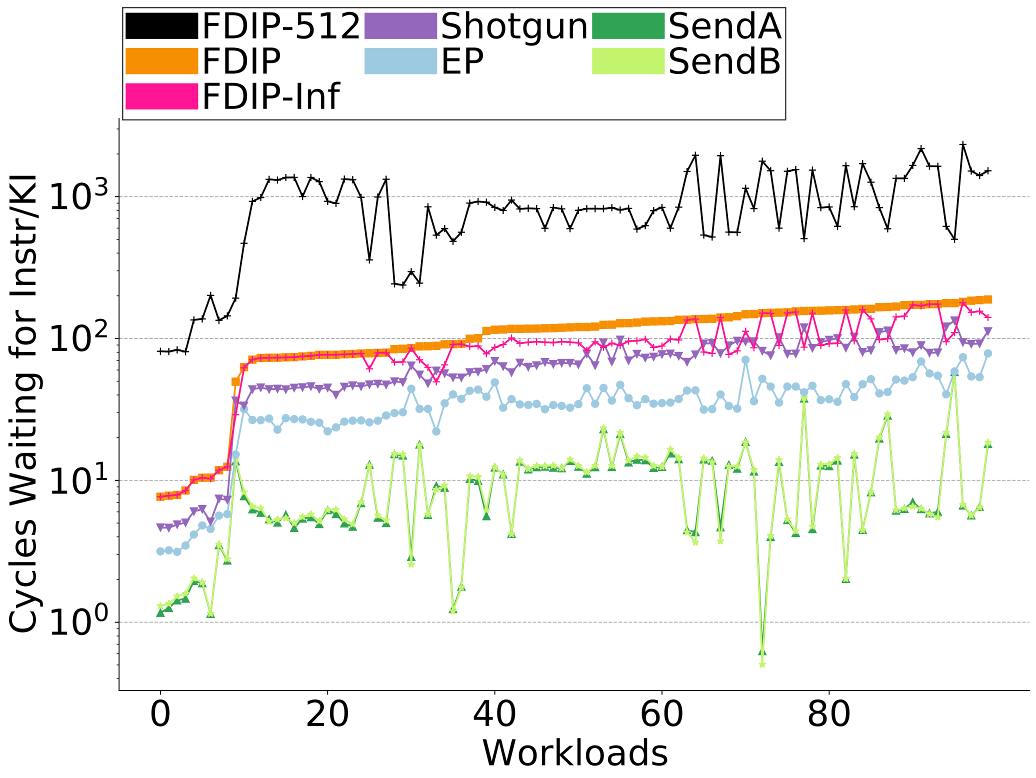


Figure 5.23: Varying Cycle Lookahead with 20-cycle L3 Latency (uA2)

Figure 5.23 quantifies the cycles spent waiting for an instruction to be fetched, expressed as cycles waiting per kilo-instructions (on a logarithmic scale), with an L3 latency of 20 cycles, using the aggressive microarchitecture. The figure shows these cycles for FDIP-512 (with a 512-entry BTB), FDIP (with

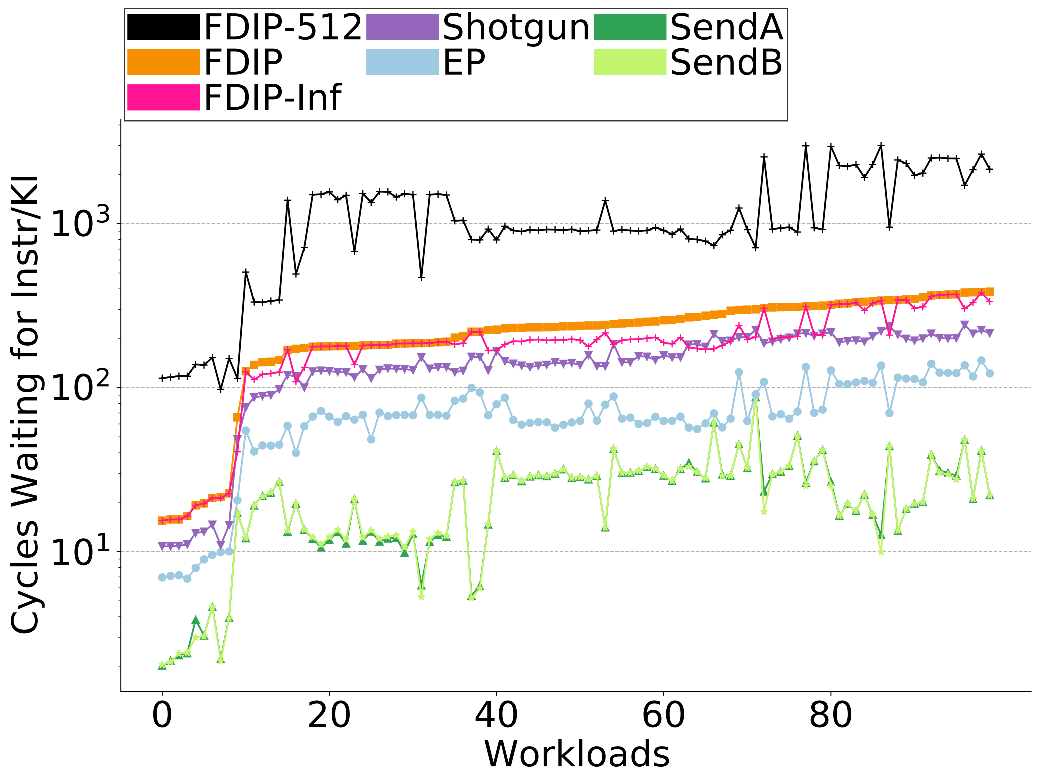


Figure 5.24: Varying Cycle Lookahead with 40-cycle L3 Latency (uA2)

an 8K-entry BTB), FDIP-Inf (FDIP with an infinite BTB), SG, EP, SendA, and SendB. The trends remain consistent with those observed earlier, with Send continuing to maintain low cycles waiting per KI even with an aggressive backend.

Similarly, figure 5.24 quantifies the cycles spent waiting for an instruction to be fetched, expressed as cycles waiting for an instruction per kilo-instructions (in log scale), for the same configurations, with a higher L3 latency of 40 cycles, using the aggressive microarchitecture. For Send, we use a configuration with a lookahead of 240 instructions. The trends remain similar to those observed earlier: cycles waiting are increased for all configurations, but Send continues to outperform the other configurations and maintain small cycles waiting in many cases. Send continues to maintain low cycles waiting with an aggressive backend and a slower L3.

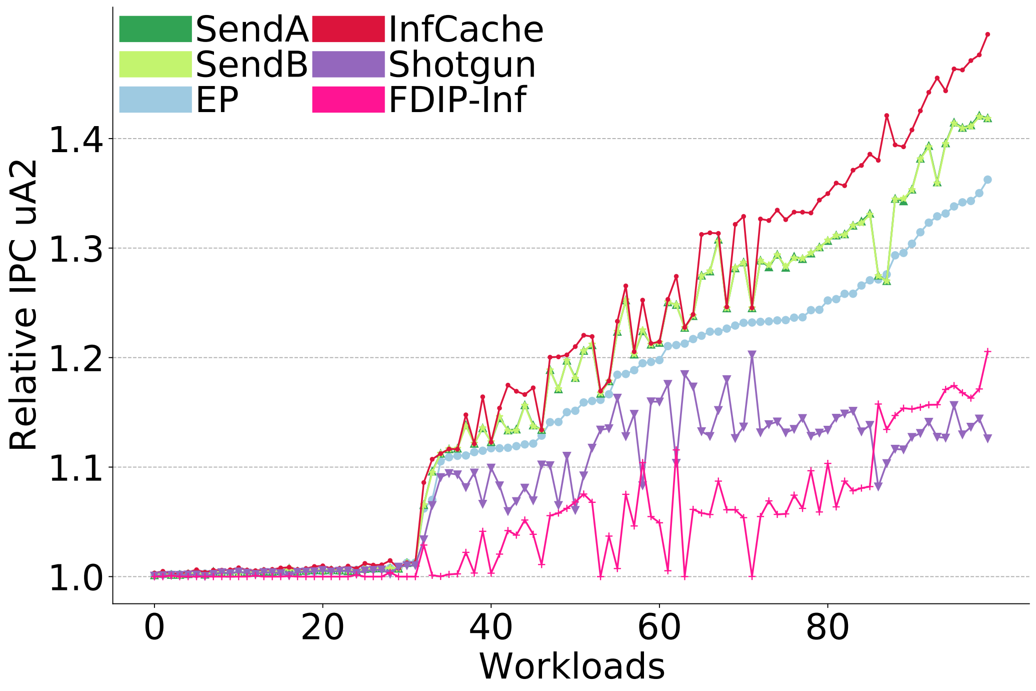


Figure 5.25: Relative IPC uA2

Reducing the cycles spent waiting for an instruction enhances the supply of instructions to the processor's backend, leading to improved performance. These performance benefits are likely to be further amplified given the more aggressive backend. Next, we quantify the increase in IPC performance.

Figure 5.25 plots the relative IPC over FDIP with an 8K BTB. We study the relative IPC of SendA, SendB, EP, Infinite L1i (InfCache), SG, and FDIP with an infinite BTB (FDIP-Inf).

SendA, SendB, and EP continue to outperform SG and FDIP-Inf by a larger margin in this more aggressive setup. The performance difference between Send and EP also becomes more noticeable. Send can keep up with the increased demands on the front end and outperforms EP in many of the

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
FDIP-8K	2.11	2.08	1.72	1.79	2.12
FDIP-Inf	2.14	2.28	1.77	1.81	2.12
Shotgun	2.16	2.32	1.84	1.89	2.32
SendA	2.21	2.66	1.92	1.92	2.37
SendB	2.20	2.66	1.92	1.92	2.37
EP	2.19	2.56	1.88	1.91	2.35
InfCache	2.23	2.75	1.93	1.92	2.38
RPKI					
FDIP-8K	5.57	7.11	5.53	4.39	5.13
FDIP-Inf	5.50	6.25	5.23	4.29	5.12
SG	5.63	7.69	5.81	4.56	5.40
Send	0.15	1.08	0.19	0.15	0.09
Cycles Waiting for Instruction Per KI					
FDIP-8K	23.48	127.32	105.33	104.23	159.26
FDIP-Inf	21.43	88.29	93.49	99.84	158.56
SG	14.23	77.50	60.39	58.19	80.96
EP	9.60	36.80	37.77	34.06	51.94
SendA	3.81	12.39	8.26	5.40	4.09
SendB	3.91	12.64	8.45	5.54	3.80

Table 5.11: Absolute IPC, RPKI and Cycles Waiting Per KI for MPKI Bins with an aggressive microarchitecture

benchmarks by 2-6%. Send comes close to the performance of InfCache on some benchmarks, there remains room for improvement on others. We next present data for the different L1i MPKI bins to better illustrate this.

Table 5.11 presents the absolute IPC, RPKI, and Cycles Waiting for an Instruction Per KI averages across different MPKI bins for various configurations, with the aggressive microarchitecture. We do not show the RPKI for EP since it is identical to FDIP-8K, as EP does not explicitly sequence the program control flow. RPKI for both SendA and SendB are also identical. The first 7 data rows show the absolute IPC for the different configurations, followed by 4 data rows presenting the RPKI for select configurations. The last 6 data rows show the Cycles Waiting for Instruction Per KI for the various configurations. The InfCache configuration has zero waiting cycles since instructions are always available to the processor. The difference in IPC between the InfCache and Send continues to be small for bins III, IV, and V because they maintain a very low RPKI, with an aggressive backend. The difference is more significant for bin II due to a higher RPKI and higher cycles spent waiting for an instruction per KI. The difference becomes quite pronounced with an aggressive backend. Many applications in bin I have a low RPKI, but a few applications have a higher RPKI, resulting in the IPC difference between InfCache and Send.

Varying Coupling with Processor

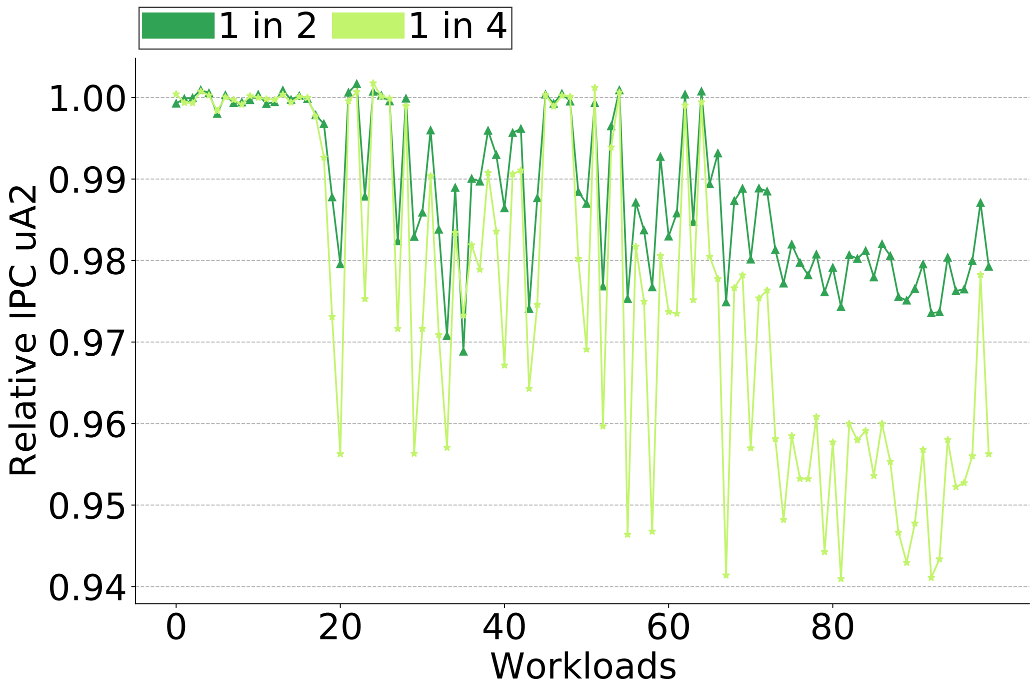


Figure 5.26: Varying Rate of Sending Fragment IDs

The IPU stays coupled to the processor using fragment identifiers to ensure it remains in sync. For the aggressive microarchitecture presented, we also

Config	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
FDIP-8K	2.11	2.08	1.72	1.79	2.12
Send-1in1	2.21	2.66	1.92	1.92	2.37
Send-1in2	2.19	2.54	1.89	1.91	2.35
Send-1in4	2.18	2.49	1.88	1.91	2.34

Table 5.12: Absolute IPC Varying Fragment ID Send Rate

study the effect of varying the rate of sending fragment identifiers to understand the impact of further reducing the coupling with the processor.

Figure 5.26 shows the IPC of Send when every 1 in 2 and 1 in 4 fragment IDs are sent from the processor to the IPU, relative to the IPC of sending every fragment ID to the IPU. For some benchmarks, the performance degradation is higher compared to others where the performance degradation is minimal.

Table 5.12 provides more insight into this aspect. We present the average absolute IPC for all MPKI bins. The average IPC is presented for FDIP (FDIP-8K) as well as Send when every fragment ID is sent from the processor to the IPU (Send-1in1) and when 1 in 2 (Send-1in2) and 1 in 4 (Send-1in4) fragment IDs are sent.

As shown in Table 5.12, there is a performance benefit with all differing rates of fragment ID sending compared to FDIP. However, for all bins except bin II, the performance degradation from S32-1in1-IPC to S32-1in4-IPC is small.

The degradation is higher for bin II because this bin has higher redirects and benefits from closer coupling to the processor. The other bins, which have fewer redirects, continue to stay on track with reduced coupling to the processor.

5.8 Creation and Updating of Fragment Table

Next, we present data to illustrate the amount of communication with the processor required to create and maintain the Fragment Table (FT), which is

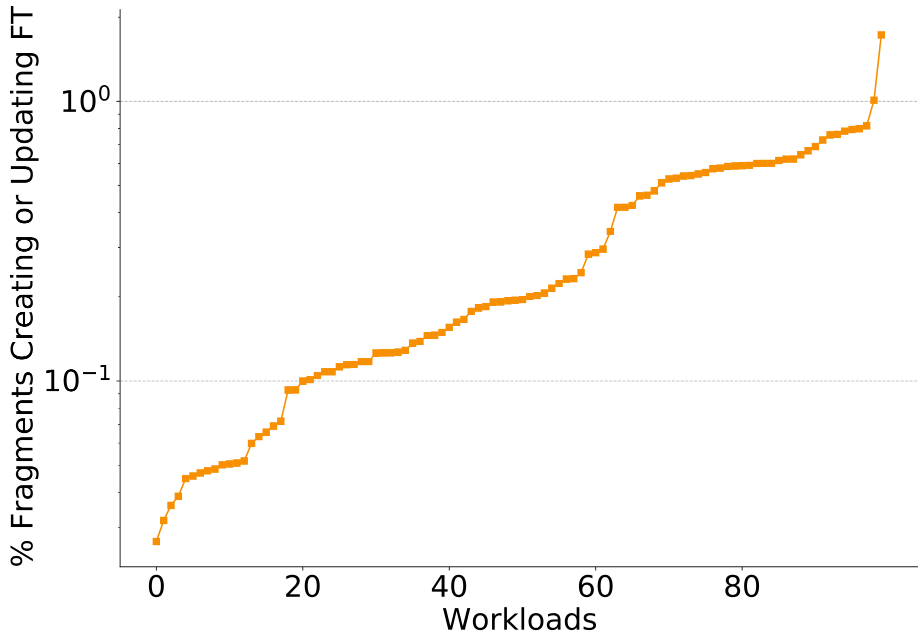


Figure 5.27: FT Creation and Updating

the key data structure used by the IPU.

Figure 5.27 shows the percentage of fragments processed (dynamic fragments) that are involved in the creation or update of the FT for all benchmarks. Note the minimal percentage for most benchmarks. This percentage is as small as 0.03% for some benchmarks meaning that only 0.03% of the dynamic fragments are involved in the creation or updating of the FT, while 99.97% of the fragments are not. This indicates that very little communication is required with the processor to create and maintain an FT.

5.9 Study with Small Cache Sizes

The program representation learned by Send is independent of the L1i size, suggesting that Send is likely to maintain a low MPKI and cycles waiting for an instruction even with smaller instruction caches, provided it can maintain

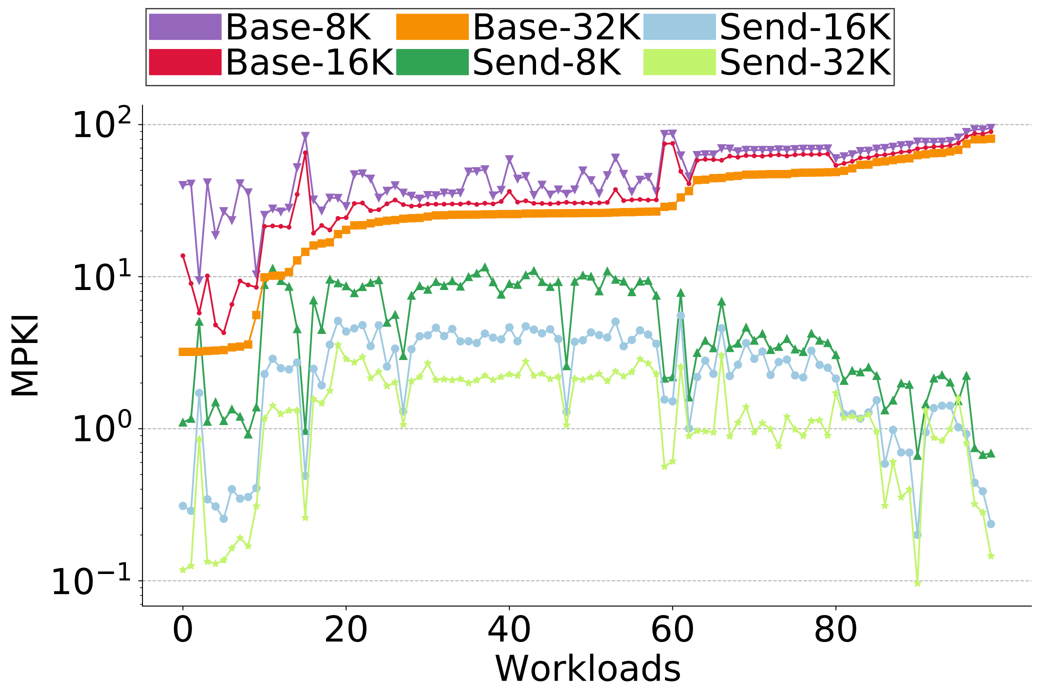


Figure 5.28: Varying L1i Size

low redirects. This section aims to evaluate this aspect.

To quantify this, we study the MPKI and cycles waiting for an instruction for Send with smaller L1i sizes in Figure 5.28. This figure plots the L1i MPKI for Base with L1i sizes of 8KB, 16KB, and 32KB (Base-8K, Base-16K, and Base-32K), and Send with L1i sizes of 8KB, 16KB, and 32KB (Send-8K, Send-16K, and Send-32K) on a log scale.

Send maintains a significantly lower MPKI even with an 8KB cache compared to all Base configurations. For applications with substantial miss reductions using Send (from nearly 100 to single digits and below 1), the Send MPKI with a 16KB cache closely approaches the MPKI of Send with a 32KB cache. In some cases, even an 8KB cache achieves an MPKI close to that of a 32KB cache.

Similarly, for some applications, Base MPKI minimally changes with the use of larger caches, in contrast to other applications where there is a more significant reduction in MPKI with larger cache sizes.

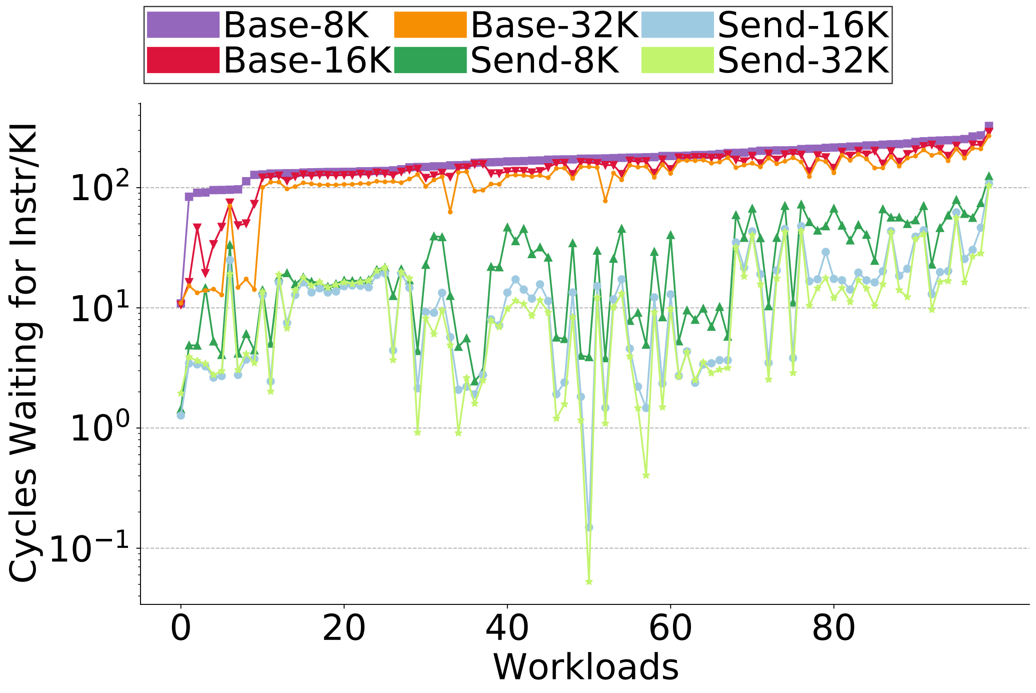


Figure 5.29: Cycles Waiting for Instruction Per KI with Small Cache Sizes

Next, we present the cycles the processor spends waiting for an instruction across all these configurations. Figure 5.29 studies the cycles waiting for instructions per KI for all of these configurations. We observe that for some benchmarks, the cycles waiting are similar for all Base configurations, while for others, Base-16K and Base-32K significantly reduce the waiting cycles. Send-8K significantly reduces the waiting cycles compared to all Base configurations, in some cases coming close to Send-16K and Send-32K. Send-16K often comes close to Send-32K. Send-32K reduces the cycles waiting to the single digits or lower, as we had seen in Figure 5.16.

Table 5.13 provides further insight into the impact of different L1i cache sizes on MPKI and Cycles Waiting for Instruction Per KI across various MPKI bins.

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
MPKI					
Base-8K	29.16	40.55	60.85	67.89	82.57
Base-16K	12.44	30.78	53.61	61.81	76.50
Base-32K	5.42	24.45	40.65	50.14	69.38
Send-8K	2.35	7.75	3.90	2.95	1.32
Send-16K	0.71	3.59	2.67	1.89	0.68
Send-32K	0.35	2.04	1.35	0.95	0.51
Cycles Waiting for Instruction Per KI					
Base-8K	109.35	196.91	164.10	153.37	174.87
Base-16K	63.52	162.55	154.03	144.36	166.31
Base-32K	34.76	145.06	127.00	125.46	154.13
Send-8K	10.89	36.86	17.33	11.90	6.09
Send-16K	6.46	16.24	12.41	7.49	1.97
Send-32K	6.42	12.86	10.99	6.99	1.33
RPKI	0.15	1.08	0.19	0.15	0.09

Table 5.13: Small Cache MPKIs

The first three data rows show the L1i MPKI (averages) for different MPKI bins with the Base configuration and cache sizes of 8KB, 16KB, and 32KB. The next three data rows show the L1i MPKI for the different bins with Send and cache sizes of 8KB, 16KB, and 32KB. Similarly, the next three data rows show the Cycles Waiting for Instruction Per KI (averages) for different MPKI bins with the Base configuration and the following three data rows shows the waiting cycles per KI with Send for the different configurations. The last data row presents the RPKI with Send for the different bins.

For bins I and II, the MPKI for Base increases by over 50% when reducing the cache size from 32KB to 8KB. Similarly, we also observe a significant increase in waiting cycles for these applications. This indicates that a 32KB cache captures a lot of temporal reuse for these benchmarks, leading to many more misses with the smaller cache size. The relative increase in MPKI is much smaller for bins III, IV, and V, suggesting less temporal reuse.

Send can maintain low MPKIs and waiting cycles per KI even with an 8KB cache for most bins. However, for bin II, the MPKI and waiting cycles per KI is relatively higher due to more frequent redirects. Similarly, Send maintains low MPKIs with a 16KB cache for all bins, indicating its effectiveness in reducing misses across different cache sizes.

5.10 Fetch FragmentIDs and Wrong Path Execution

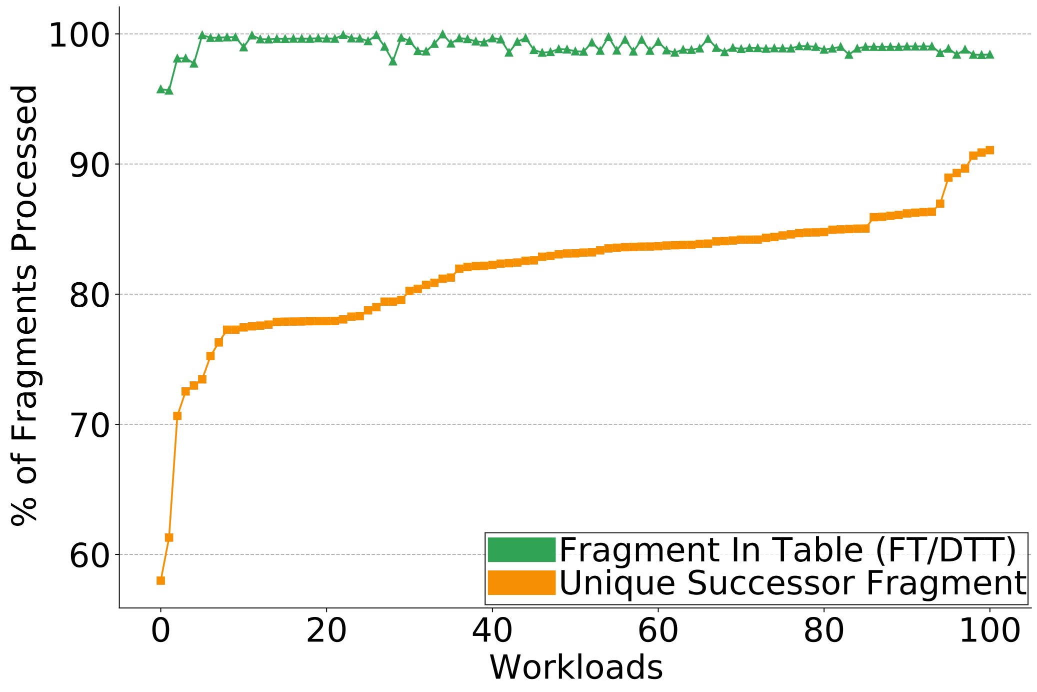


Figure 5.30: Fragment Level Control (Wrong Path)

In this section, we study some properties of dynamic fragment-level control flow to understand how redirects are likely to be affected in the presence of wrong-path execution.

The Unique Successor Fragment curve in Figure 5.30 measures the percentage of dynamic fragments that contain a branch misprediction yet still have a unique successor fragment. This percentage exceeds 80% for many benchmarks, indicating that fragment-level control flow remains unaffected by branch mispredictions within a fragment more than 80% of the time.

The Fragment In Table (FT/DTT) curve in Figure 5.30 measures the percentage of dynamic fragments containing a mispredicted branch where the successor fragment is found either in the FT or the DTT. For most benchmarks, this percentage is around 99%. This indicates that even in the presence of a branch misprediction, the IPU can operate without experiencing a redirect by pursuing upto two tracks for approximately 99% of the fragments, using the Fetch FragmentID to stay synchronized with the processor.

5.11 Using Retire Fragment Identifiers

The IPU stays synchronized with the processor using fragment IDs. Throughout our evaluation, we have used fetch fragment IDs. Using fetch fragment IDs offers the advantage of providing early feedback to the IPU if it is on an incorrect path, allowing it to quickly correct. However, many of the benchmarks we study have a low frequency of redirects, suggesting that they could perform well even if this feedback comes later in the pipeline. Retire fragment IDs have the advantage of not being affected by wrong-path execution, thus avoiding redirections caused due to wrong-path execution. However, retire fragment IDs provide feedback to the IPU much later in the instruction pipeline, delaying corrections. Given the low redirect rates for many benchmarks, we expect the impact of using retire fragment IDs on performance to be minimal, which we will evaluate next.

Figure 5.31 shows the relative IPC over FDIP with an 8K BTB for Send using fetch fragment IDs and retire fragment IDs. This evaluation is carried out using the aggressive microarchitecture detailed in Section 5.7.1. Although the use of retire fragment IDs still performs better than FDIP, it clearly results in slowdowns compared to using fetch fragment IDs. This is not surprising, as

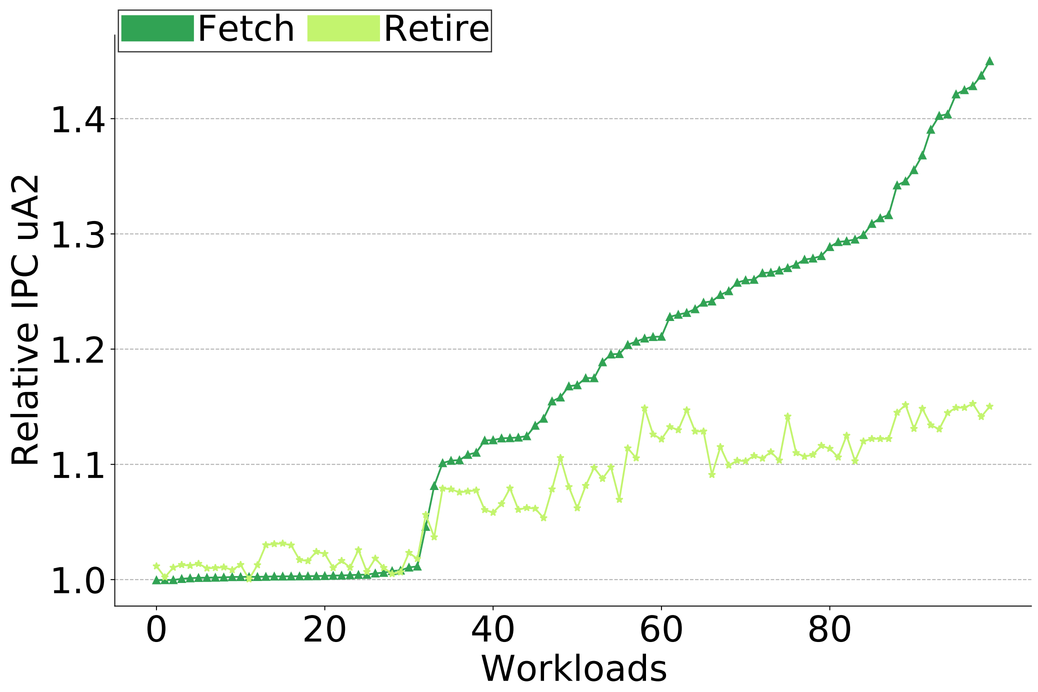


Figure 5.31: Relative IPC Retire Fragment IDs

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
% Slowdown	1.66	12.57	5.80	2.39	3.65
RPKI	0.15	1.08	0.19	0.15	0.09

Table 5.14: Performance Degradation with Retire Fragments for MPKI Bins

the IPU receives feedback that it is on an incorrect path much later than when using fetch fragment IDs. In some cases, the slowdowns are significantly higher. When redirects are more frequent, the slowdowns are expected to be greater than when redirects are less frequent. We study this aspect next.

Row 1 in Table 5.14 shows the average slowdowns associated with the usage of retire fragments compared to the use of fetch fragment identifiers for all MPKI bins. Additionally, row 2 in the table presents the RPKI values for the different MPKI bins. Bin II has a high RPKI and also experiences more than a

10% slowdown in IPC performance, compared to other bins that have much lower RPKI and relatively smaller slowdowns when using retire fragment identifiers, as expected.

A very low rate of redirects opens up the possibility of using retire fragment identifiers in place of fetch fragment identifiers.

5.12 Summary

In this chapter, we first presented the experimental infrastructure and setup used for evaluating Send.

We quantified that for many applications, Send can sequence the program at a high level with near-perfect accuracy, maintaining low rates of redirects. We examined high-level sequencing in detail, focusing on how it is influenced by the scheme's operation and the size of key data structures. This sequencing can effectively identify the blocks of code likely to be referenced by the processor. Next, we quantified the miss reductions achieved by Send, stemming from this sequencing and block identification, which significantly reduces misses for many benchmarks. We also analyzed the block movement traffic introduced by Send in detail.

The significantly reduced misses with Send resulted in the processor spending very little time waiting for instructions. We studied in great detail this metric and also observed that Send continues to maintain a small cycles waiting for an instruction when the L3 is made slower. Next we presented the overall IPC performance improvement for Send and related schemes. As a result of maintaining a small cycles waiting for an instruction, its performance often approached that of an infinite cache. Furthermore, Send continued to maintain small cycles waiting for an instruction and deliver high performance close to that of an infinite cache, even as the backend was made more aggressive.

Additionally, we studied other aspects of the design, such as the Block Temperature Table (BTT), which can be used to control block movement, and the Pseudo Inclusion Table (PIT), which can be used to reduce L1i tag probes. Moreover, we examined the impact of using retire fragment IDs, the effect of wrong-path execution on redirects, and the implications of working with smaller L1is.

Chapter 6

Presending for an iTLB

Instruction Presending, as described in the last two chapters, creates a shadow program representation that is used to proactively move code cache blocks from a lower-level cache to an L1i. Given that instruction TLB entries are closely related to the static code cache blocks accessed, this program representation can potentially be enhanced to support the proactive movement of TLB entries from an L2 TLB to an L1 TLB (iTLB).

First, we describe the additions to the program representation created earlier for the movement of code cache blocks. Next, we outline the necessary changes to Presending to support the movement of TLB entries. Finally, we evaluate the movement of TLB entries with this functional enhancement.

6.1 Program Representation Enhancements

The program representation facilitates sequencing fragment-level control flow to identify the blocks of instructions the processor is likely to reference. Additionally, this representation can be used to identify the VPN-PTEs the processor is likely to reference. We describe the additions to the program representation next.

6.1.1 Representing TLB Entries for a Fragment

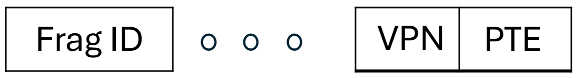


Figure 6.1: Code Region for TLB

In many cases, there is likely to be spatial locality in the blocks accessed by a fragment, resulting in accesses to contiguous blocks of code or code regions, as discussed in the previous chapter. Code regions are encoded as a block and a count of blocks to identify the blocks accessed. In addition, they could be encoded using a Virtual Page Number (VPN) and Page Table Entry (PTE) pair to encode the translation accessed by the code region. An example encoding for a code region accessed by a fragment is shown in Figure 6.1.

Alternatively, a pointer to the L2 TLB entry could be stored in place of the VPN-PTE pair as a space optimization.

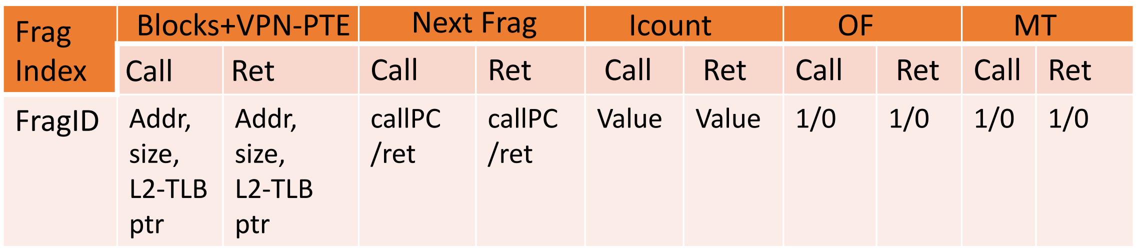
Separate (discontiguous) code regions can be handled similarly to code regions for blocks, with the VPN-PTEs for the separate regions stored separately.

While storing the pointer to the L2 TLB results in space optimization, evictions of L2 TLB entries would result in the program representation containing pointers to incorrect TLB entries. This would cause incorrect TLB entries to be moved to the iTLB, which has no functional implication. It would lead to subsequent iTLB misses, at which point the pointer can be updated to the new location.

Further, processor designs such as AMD's Zen3 [28] are moving towards a split L2 TLB between data and instructions, making evictions in the L2 TLB less frequent. To avoid performance pathologies, we can periodically clear the VPN-PTEs in the representation and relearn them on subsequent iTLB misses.

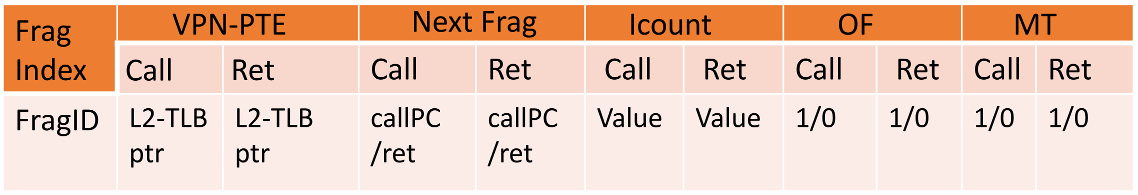
Any changes to the VPN-PTEs can simply clear all the VPN-PTEs in the representation, resulting in relearning the VPN-PTEs accessed by the different fragments on subsequent iTLB misses.

6.2 Hardware and Operational Enhancements for TLB Entry Movement



Frag Index	Blocks+VPN-PTE		Next Frag		Icount		OF		MT	
	Call	Ret	Call	Ret	Call	Ret	Call	Ret	Call	Ret
FragID	Addr, size, L2-TLB ptr	Addr, size, L2-TLB ptr	callPC /ret	callPC /ret	Value	Value	1/0	1/0	1/0	1/0

Figure 6.2: Entry of a Fragment Table for Blocks and TLB Entry Movement



Frag Index	VPN-PTE		Next Frag		Icount		OF		MT	
	Call	Ret	Call	Ret	Call	Ret	Call	Ret	Call	Ret
FragID	L2-TLB ptr	L2-TLB ptr	callPC /ret	callPC /ret	Value	Value	1/0	1/0	1/0	1/0

Figure 6.3: Entry of a Fragment Table for only TLB Entry Movement

Figure 6.2 shows an entry in the FT entry with all components integrated, where the code region in every entry is augmented with a pointer to an L2 TLB entry to facilitate the movement of corresponding VPN-PTEs for the code regions from a lower-level TLB to the L1 TLB. L2 TLB pointers are also added to the code regions stored in the Overflow Regions Table (ORT).

Alternatively, if the FT is used to only identify the VPN-PTEs likely to be referenced by the processor, it can hold only the VPN-PTEs accessed by the

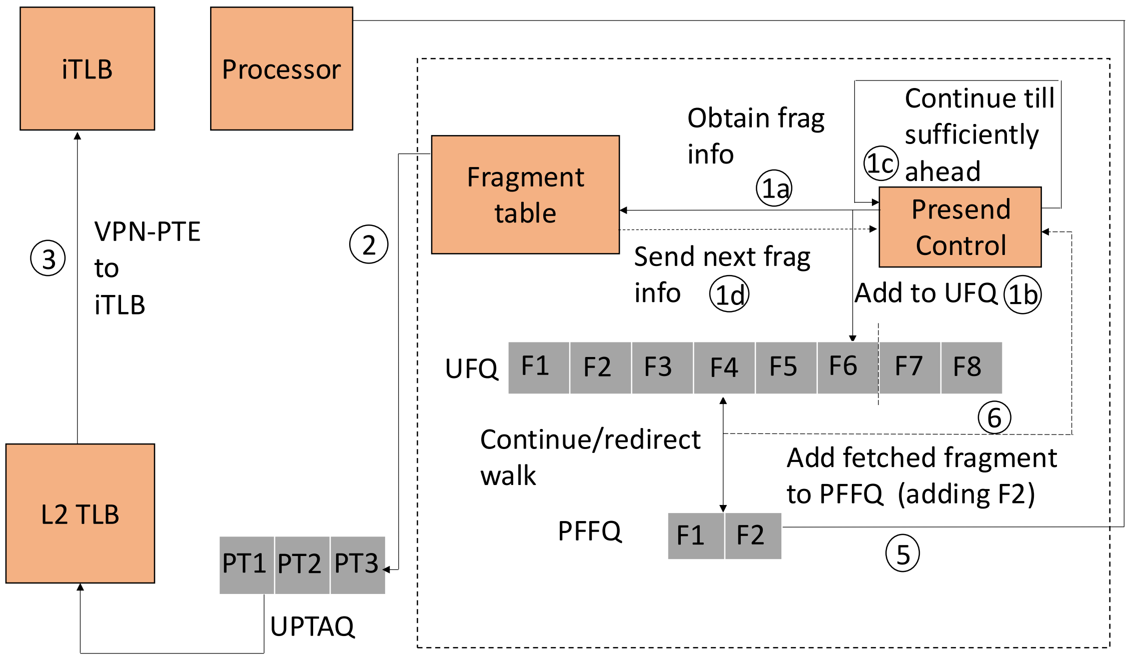


Figure 6.4: Instruction Presending Unit (IPU) operation for an iTLB

fragments without storing the code block regions, as shown in Figure 6.3. This approach can result in storage savings, if important.

The pointers of the L2 TLB entries in that fragment are placed into an Upcoming Page Table Addresses Queue (UPTAQ), as shown in Figure 6.4. For entries in the UPTAQ, a decision is made (similar to blocks, to reduce the movement of VPN-PTEs into the L1 TLB) whether to read a TLB entry from the L2 TLB and send it to the L1 TLB (iTLB).

6.3 Evaluation

All the parameters used for the simulation are the same as those used in the previous chapter for the evaluation of Instruction Presending for code blocks. All evaluations are performed with a unified 3K L2 TLB, which is reasonable given the increase in L2 TLB sizes, such as the 3K L2 DTLB in the AMD Zen 4 microarchitecture [15].

An additional 13.5KB of storage is required on top of the existing 68KB to encode the L2 TLB pointers (1.5B each) for all the code regions accessed by different fragments. This facilitates the identification of both code blocks and VPN-PTEs for the various fragments. Alternatively, representing only VPN-PTEs via L2 TLB pointers, to facilitate the identification of only the VPN-PTEs likely to be referenced, results in a total storage requirement of 49KB.

For Instruction Presending (Send), we only evaluate the movement of TLB entries in this Chapter.

Next we provide a brief outline of the results we present:

- We primarily focus on evaluating the movement of TLB entries and the associated overheads, as this is the key functional enhancement introduced.
- We study the overall IPC performance with Send and provide comparisons against a large iTLB and an infinite iTLB.

Next, we describe some of the key metrics used for our evaluation.

- **MPKI** is used to better demonstrate the magnitude of miss reduction, similar to the evaluation of block movement.
- **Traffic/L2 TLB Accesses per KI** Movement of iTLB entries involve an L2 TLB access. So we use L2 TLB Accesses per KI to better illustrate the increase in TLB entry movement resulting from the operation of the scheme, similar to the evaluation of block movement.

6.3.1 Miss Reductions

Figure 6.5 presents the iTLB MPKI (log-scale). The results are shown for the baseline microarchitecture (with no TLB prefetching) with a 64-entry,

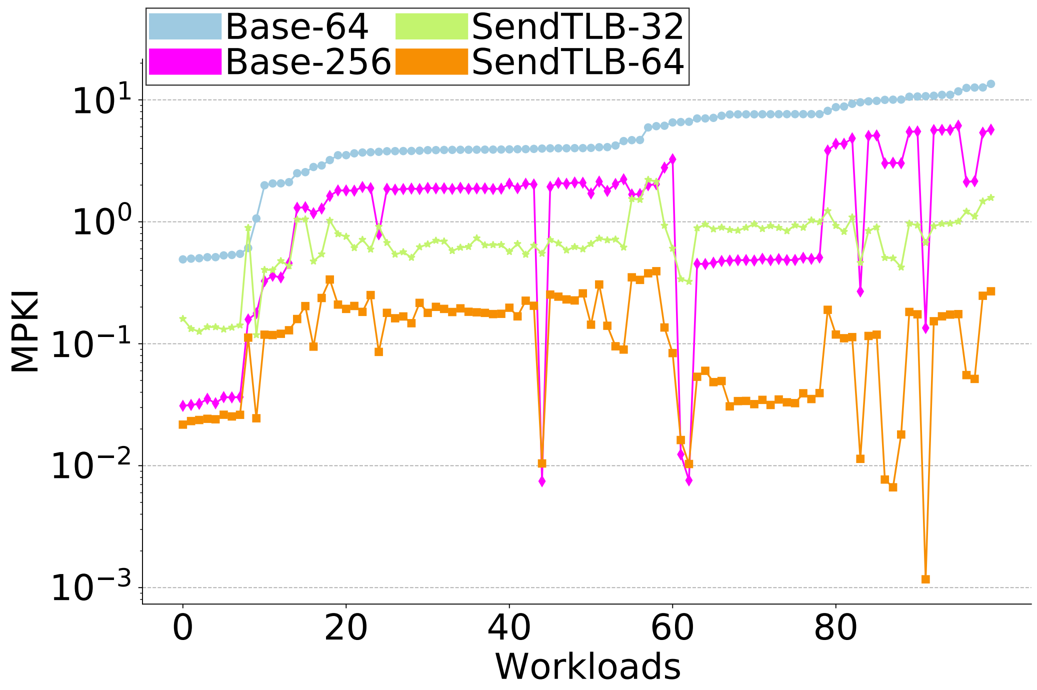


Figure 6.5: iTLB MPKI

and a 256-entry iTLB (Base-64, Base-256), and for Send with a 32-entry and a 64-entry iTLB (SendTLB-32 and SendTLB-64). SendTLB-64 significantly outperforms Base-64 and Base-256 in all cases. SendTLB-32 outperforms Base-64 and outperforms or comes close to Base-256 in many cases. These results demonstrate that Send can effectively identify and move iTLB entries to a small-sized iTLB bringing down the misses over a normal iTLB without prefetching by one or two orders of magnitude in many cases.

Next, we evaluate the impact on misses when using a Block Temperature Table (BTT) to reduce the movement of VPN-PTEs, instead of blocks, as shown in Figure 6.6. The BTT is indexed using the lower n bits of the virtual page address and has 2K entries. We present the iTLB MPKI (log-scale) for Base-64 and for Send with a 64-entry iTLB, both with and without a BTT (SendB and SendA, respectively).

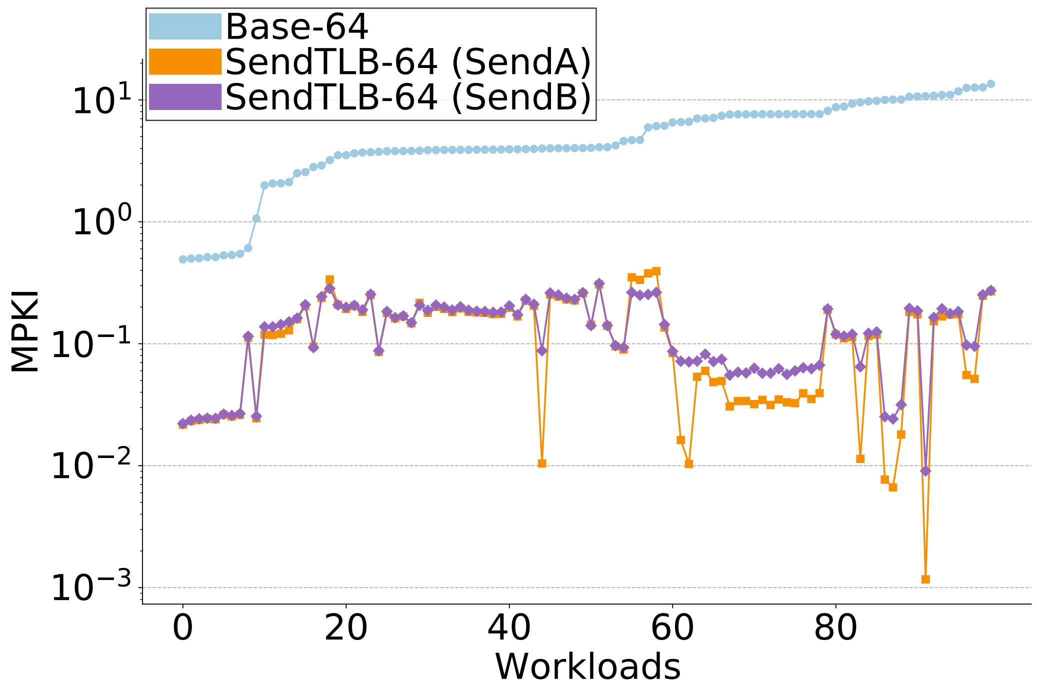


Figure 6.6: iTLB MPKI with BTT

Both SendA and SendB are nearly equally effective in reducing misses compared to Base-64. However, SendB performs slightly worse in a few cases because SendA unconditionally moves VPN-PTEs, whereas SendB may occasionally make incorrect decisions to reduce movement, resulting in slightly higher misses.

6.3.2 TLB Entry Movement Traffic

Next, we quantify the L2 TLB accesses per KI to provide an idea of how many extra translations are installed in the iTLB by Send with a 64-entry TLB, both with and without a BTT (SendB and SendA). Figure 6.7 quantifies the TLB entry movement for Base-64 and Send-64. For some of the benchmarks, we see a relatively higher increase in TLB entry movement, for some other applications, the relative increase in traffic with Send is small. We present more

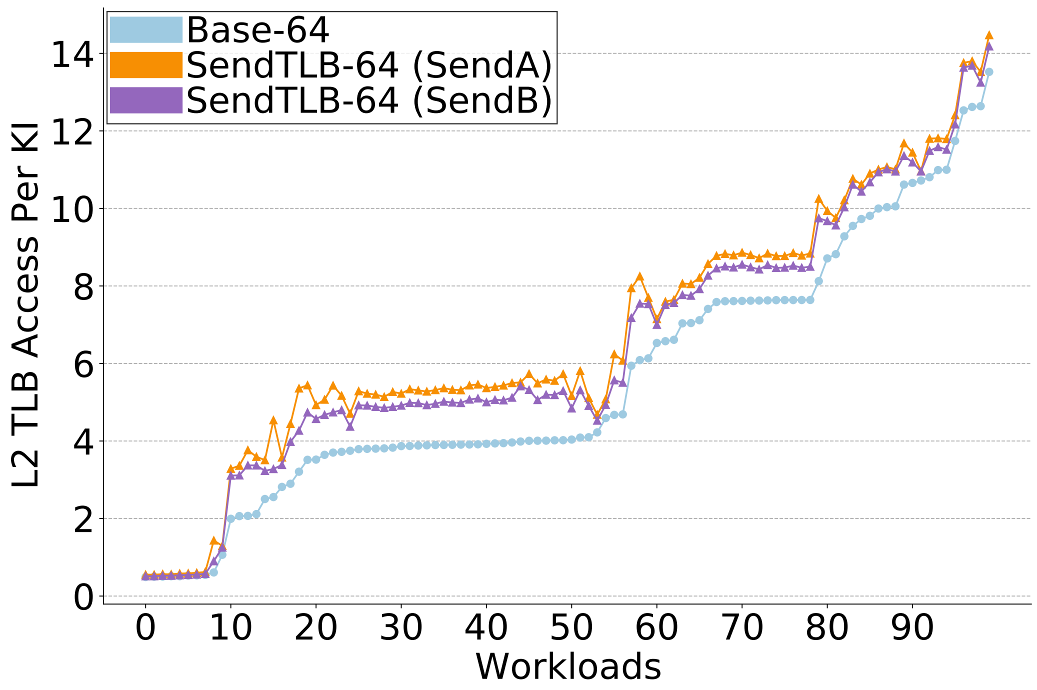


Figure 6.7: iTLB Traffic

Metric	**(I)**	**(II)**	**(III)**	**(IV)**	**(V)**
Base-64	0.99 (1.0)	3.96 (1.0)	6.97 (1.0)	8.47 (1.0)	11.23 (1.0)
SendTLB-64 (SendA)	1.32 (1.33)	5.41 (1.36)	8.18 (1.17)	9.59 (1.13)	12.16 (1.08)
SendTLB-64 (SendB)	1.21 (1.22)	4.99 (1.26)	7.91 (1.13)	9.3 (1.09)	12.02 (1.07)

Table 6.1: L2 TLB Accesses per KI for L1i MPKI Bins

data next to provide a better understanding. Further, we observe that SendB is able to bring down the TLB entry movement over SendA in many applications.

Table 6.1 presents the average L2 TLB accesses per KI for the different L1i MPKI bins. For each of the fields in these rows, the relative increase in traffic over Base-64 is shown in parentheses. We observe that the L2 TLB Accesses per KI for Base-64 is highest for bin V and lowest for bin I. The relative increase in L2 TLB accesses for both SendA and SendB is higher for bins I and

II compared to bins III, IV, and V. However, the total magnitude of L2 TLB accesses with Send is smaller compared to the other bins.

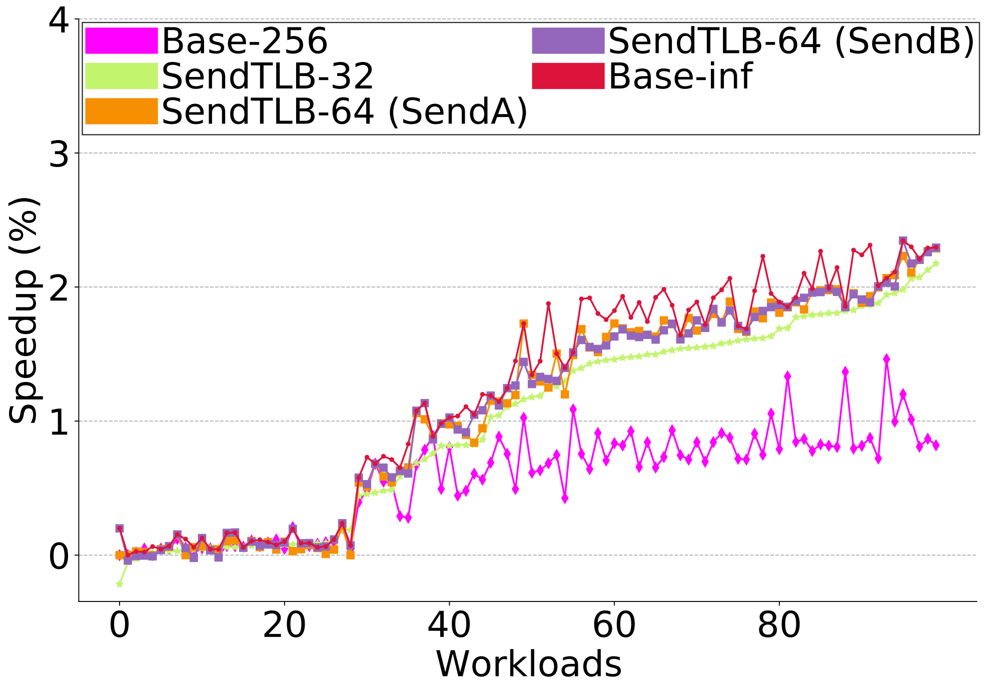


Figure 6.8: iTLB Speedup

6.3.3 IPC Performance Improvements

Reduction of iTLB misses results in an improved IPC performance which we quantify next. We plot the IPC performance speedups of SendTLB-32, SendTLB-64 (both SendA and SendB), Base-256, and Base with an infinite-entry iTLB (Base-inf) over Base-64 in Figure 6.8. Both SendTLB-32 and SendTLB-64 come close to the performance of Base-inf, outperforming Base-256 on most workloads. Speedups for SendA and SendB are mostly identical. This illustrates that Send can operate quite effectively with a small-sized iTLB, approaching the performance provided by an infinite-sized iTLB. These results mostly follow from what we observed in Section 6.3.1, where we saw that Send comes quite close to an infinite TLB in most cases.

6.4 Summary

This chapter proposed Presending for iTLBs. We described the enhancements to Presending to facilitate this movement and evaluated the supply of iTLB entries to the L1 TLB (iTLB).

From the above results, we clearly saw that Presending can effectively identify the iTLB entries likely to be referenced by the processor and keep a small-sized iTLB supplied with the necessary iTLB entries.

Chapter 7

Presending for a BTB

Instruction Presending, as described in the last three chapters, creates a shadow program representation that proactively moves code cache blocks and instruction TLB entries from a lower-level cache or L2 TLB to an L1i or iTLB. BTB entries are used to correctly fetch instructions past previously taken branches, jumps, and calls, and are closely related to the static instructions or static code cache blocks. This suggests that the Instruction Presending scheme could potentially be enhanced to support the movement of BTB entries from an L2 BTB to an L1 BTB.

First, we describe the operational enhancements to the Presending scheme to support the movement of BTB entries. Finally, we evaluate the movement of BTB entries with this functional enhancement.

7.1 Presending Operational Enhancement for BTB

Identifying BTB entries requires no additions to the program representation. Information used to identify the VPN-PTEs as well as code cache blocks accessed by fragments can collectively be used to identify the BTB entries likely to be referenced by the processor.

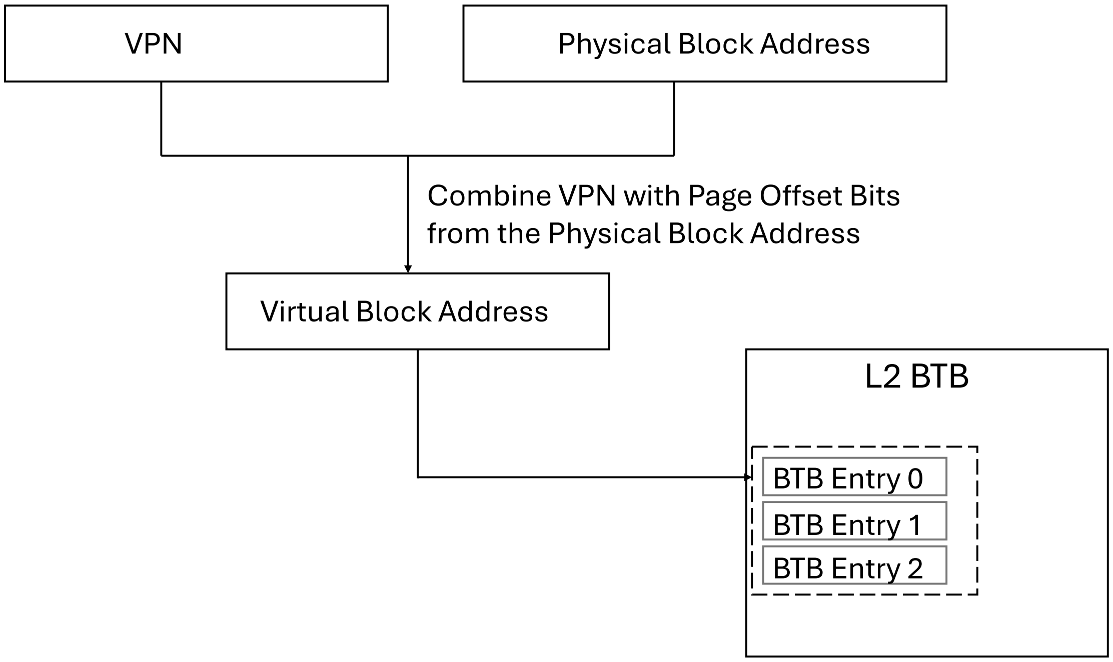


Figure 7.1: Accessing L2 BTB for BTB Entry Movement

BTBs are generally accessed using virtual addresses. To facilitate the movement of BTB entries from an L2 BTB to an L1 BTB, as shown in Figure 7.1, the IPU needs virtual addresses. Virtual page numbers can be extracted from the VPN-PTE entries corresponding to the different code regions in the FT/ORT, as is typically required for the movement of iTLB entries. Similarly, physical block addresses can be extracted for the code regions from the FT/ORT, as is typically required for the movement of cache blocks.

The virtual page number for a code region can be combined with the page offset for a given physical block address of the code region. The virtual block address generated using this process can then be used to query the BTB. In the case shown in Figure 7.1, BTB Entries 0-2 are part of the given virtual cache block address. All of these entries are accessed (over multiple BTB accesses) and copied to the L1 BTB. This process can be repeated for all cache blocks that are part of the code region. Presence checks, similar to those used for cache blocks, can be performed to reduce unnecessary movement.

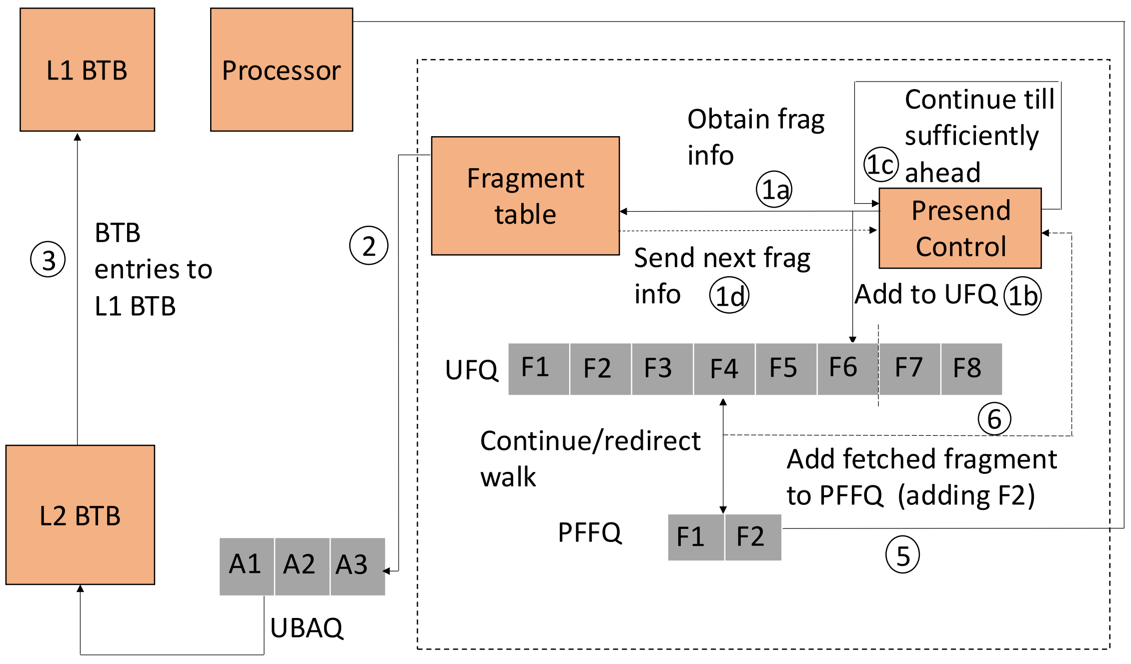


Figure 7.2: Instruction Presending Unit (IPU) operation for BTB

Operationally, this process involves placing the generated virtual block addresses in a queue (UBAQ) as shown in Figure 7.2 during the IPU's operation. For entries in the UBAQ, a decision is made (similar to blocks and iTLB entries) on whether to read BTB entries that are part of the given block and send them to the L1 BTB.

7.1.1 Working without an L2 BTB

In some implementations, an L2 BTB might not be present. In such cases, most BTB entries can be derived from the instructions within the corresponding code cache blocks through pre-decoding. The branch type and the target address are encoded in the static instruction within the code cache block.

Pre-decoding can be utilized to extract the target addresses for most control instructions (except for indirect jumps and indirect calls) to create BTB entries for installation in the L1 BTB.

7.2 Evaluation

All the parameters used for the simulation are the same as those used in the previous chapter for the evaluation of Instruction Presending for code blocks and iTLB entries. All evaluations are performed with a 16K-entry L2 BTB, which is typical of the L2 BTB sizes used in processors such as the Samsung M3 [66].

For Instruction Presending (Send), we only evaluate the movement of BTB entries in this Chapter, except the last plot where we evaluate the ability of Send to move both blocks and BTB entries simultaneously. For Shotgun, we use a configuration with a 4K U-BTB and a 1K-entry C-BTB, amounting to a total storage of approximately 64KB, same as the configuration used in Chapter 5.

- We primarily focus on evaluating the movement of BTB entries and the associated overheads for Instruction Presending, as this is the key operational enhancement introduced.
- IPC performance which shows that Send performs as well as the baseline microarchitecture with an infinite BTB.

7.3 BTB Entry Movement

7.3.1 Key Metrics

- **MPKI** is used to better illustrate the magnitude of miss reduction, similar to its use in the evaluation of block movement and TLB entry movement.

– **BTB Fills Per KI (FPKI)** is slightly different from the metric used for block movement because the baseline operation for a BTB does not involve an L2 BTB. The movement of BTB entries involves the installation of entries in the L1 BTB, so we use this metric instead. This metric is used to better illustrate the magnitude of increase in BTB installs in the L1 BTB resulting from the operation of the scheme.

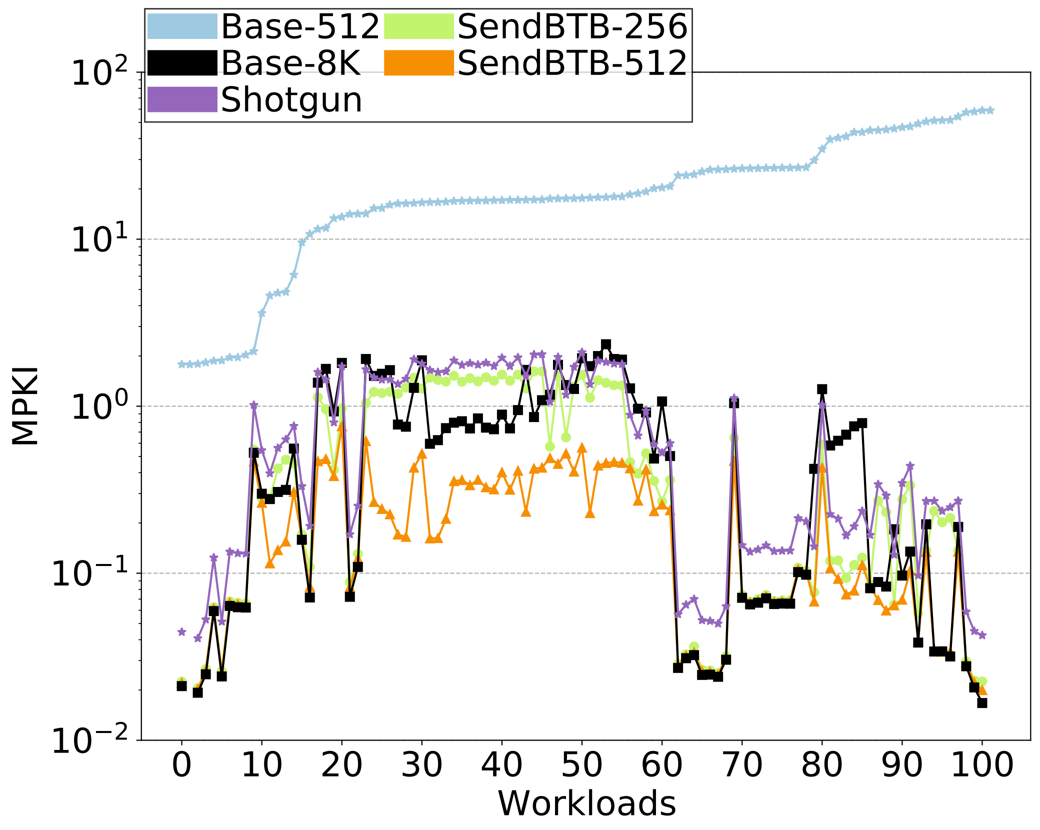


Figure 7.3: BTB MPKI

7.3.2 Miss Reductions

Figure 7.3 presents BTB MPKI (log-scale). The results are shown for the baseline microarchitecture with a 512-entry, 8K-entry L1 BTB (Base-512, Base-

8K), which do not have any BTB prefetching; for Shotgun (SG); and for Send with a 256-entry and 512-entry L1 BTB (SendBTB-256 and SendBTB-512).

Base-512 falls short due to a lack of capacity to hold targets for active control instructions.Base-8K significantly brings down the BTB MPKI. Send-256 outperforms Base-8K in many cases. Send-512 brings down the misses even further, outperforming Base-8K in some cases. Shotgun performs similarly to Base-8K and slightly better for some benchmarks.

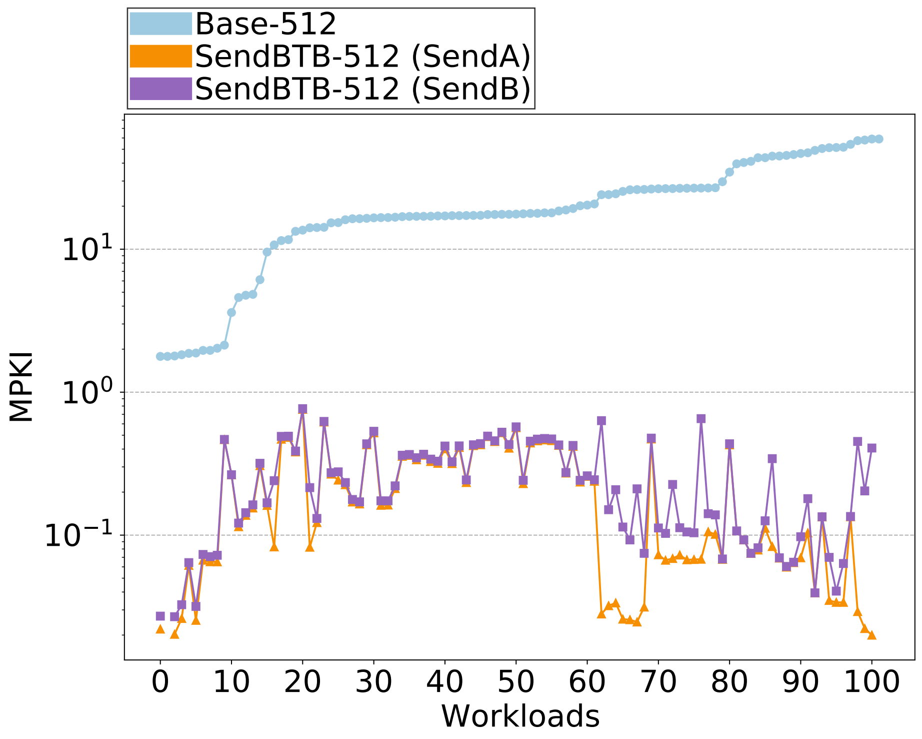


Figure 7.4: BTB MPKI with BTT

Next, we evaluate the effect on misses of using a Block Temperature Table (BTT), as shown in Figure 7.4, to reduce the movement of BTB entries. The BTT is indexed using the lower n bits of the block address and has 2K entries. We present the BTB MPKI (log-scale) for Base-512 and for Send with a 512-entry BTB, both with and without a BTT (SendB and SendA, respectively).

Both SendA and SendB are highly effective at reducing misses compared to Base-512 without BTB prefetching. In some cases, SendA performs better than SendB because the BTT can sometimes make incorrect decisions to reduce BTB entry movement, resulting in increased misses for some benchmarks.

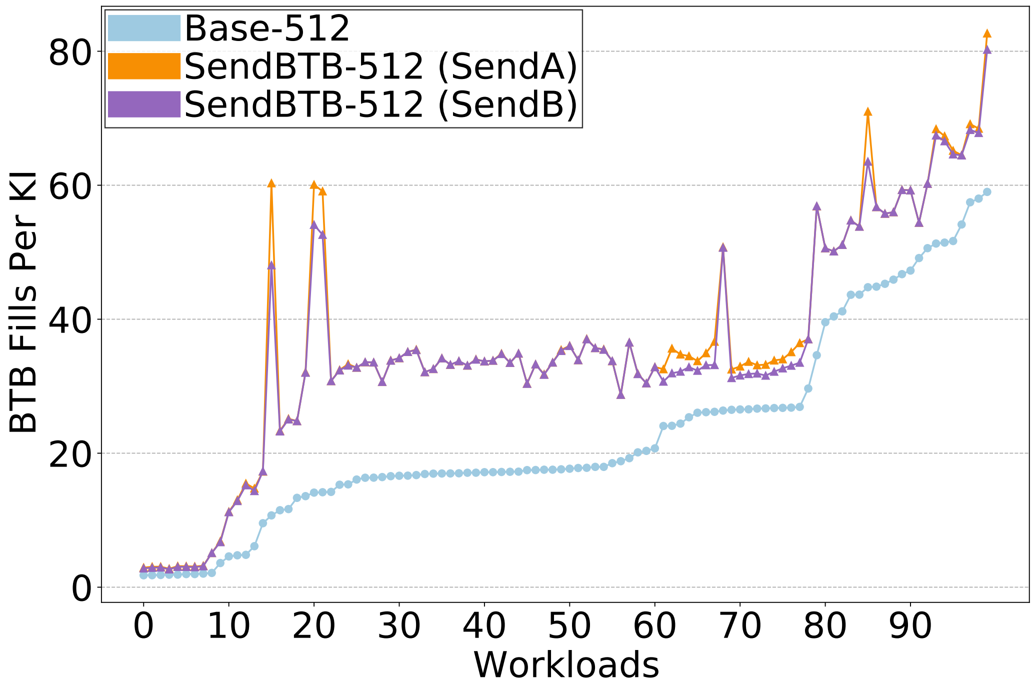


Figure 7.5: BTB FPKI

7.3.3 BTB Fills

We quantify the BTB fills per KI to illustrate the number of extra BTB entries installed in the L1 BTB by Send. Figure 7.5 quantifies the number of BTB fills per KI for Base-512 and SendBTB-512 with and without a BTT (SendB and SendA). We observe that SendB is able to bring down the BTB entry movement over SendA in many applications. We present more data to provide more insights into the added BTB entry movement.

Metric	(I)	(II)	(III)	(IV)	(V)
FPKI					
Base-512	3.12 (1.0)	16.65 (1.0)	26.95 (1.0)	32.39 (1.0)	52.08 (1.0)
SendBTB-512 (SendA)	6.45 (2.06)	33.57 (2.01)	40.34 (1.49)	41.28 (1.27)	66.05 (1.26)
SendBTB-512 (SendB)	6.22 (1.99)	33.39 (2.00)	38.72 (1.20)	39.97 (1.23)	64.88 (1.25)
MPKI					
Base-512	3.12	16.65	26.95	32.39	52.08
SendBTB-512 (SendA)	0.08	0.33	0.09	0.06	0.05
SendBTB-512 (SendB)	0.10	0.34	0.26	0.11	0.14

Table 7.1: BTB FPKI and MPKI for L1i MPKI Bins

First three data rows in Table 7.1 presents the average BTB Fills Per KI for the different L1i MPKI bins. First data row quantifies the BTB entry movement for Base-512 and the second and third data rows quantifies the BTB entry movement for SendBTB-512. For each of the fields in these rows, the relative increase in traffic over Base-512 is shown in parentheses. BTB entry movement for Base-512 is lowest for bin I and is highest for bin V. Relative increase in BTB entry movement is highest for bin I and bin II (over 100%) for both configurations of Send, though the magnitude is smaller compared to the other bins. Relative increase in BTB entry movement is smaller for bins IV and V, where there is higher BTB entry movement to begin with.

The last three data rows in Table 7.1 present the average BTB MPKI across the different L1i MPKI bins. Send is effective in reducing BTB MPKI across all bins, though bin II shows slightly higher BTB MPKI. We also observe that BTB MPKI is higher for the bins with SendB, as the use of BTT can occasionally lead to incorrect decisions.

7.3.4 IPC Performance Improvement

Given that presending can significantly reduce BTB misses even with a small-sized L1 BTB, we next evaluate the IPC performance of Send with small-sized L1 BTBs. This evaluation aims to understand how closely it approaches the performance of the Base configurations, particularly those with larger L1

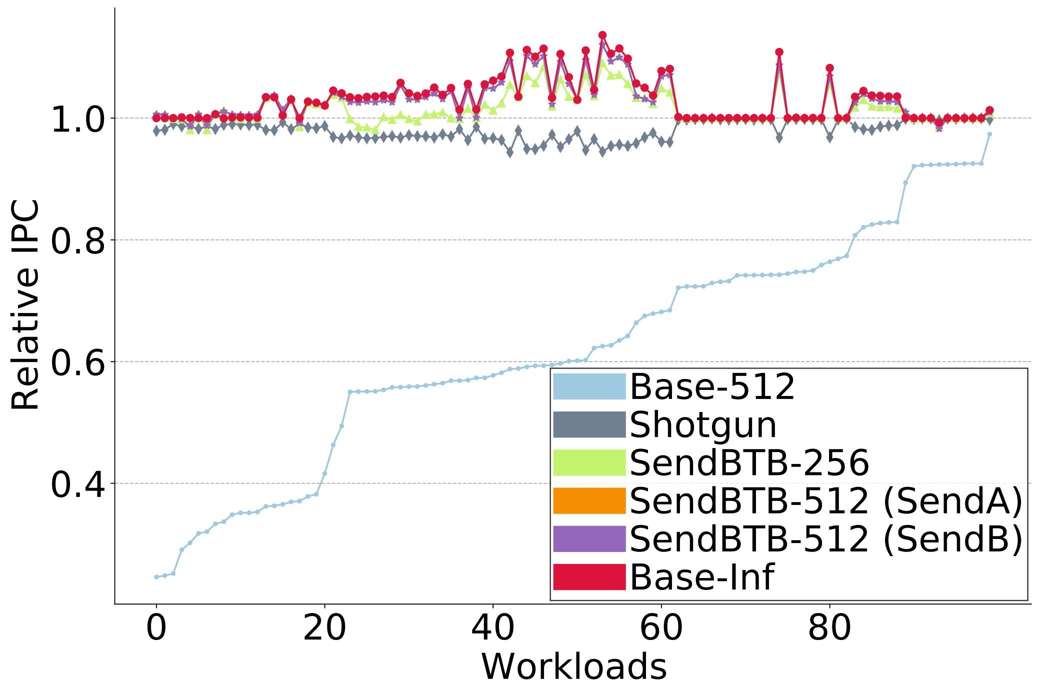


Figure 7.6: BTB IPC

BTBs (an 8K-entry BTB and an infinite BTB). Figure 7.6 presents the relative IPC of all configurations with respect to Base-8K. We also present the relative IPC for a baseline with an infinite BTB (Base-Inf). Base-512 has an IPC significantly lower than Base-8K. This is because of the significantly higher BTB misses with Base-512 compared to Base-8K, as we saw in Section 7.3.2. The baseline microarchitecture simulated is FDIP and frequent BTB misses result in a significant slowdown. SG performs quite close to Base-8K, as its BTB MPKI is very similar to Base-8K, as we saw in Section 7.3.2. Send achieves better IPC than Base-8K for many applications with just 256 BTB entries. With 512 BTB entries, Send comes close to the performance of an infinite BTB for most benchmarks studied. The relative IPC for SendA and SendB are mostly identical.

7.3.5 IPC Performance with BTB+Block Presending

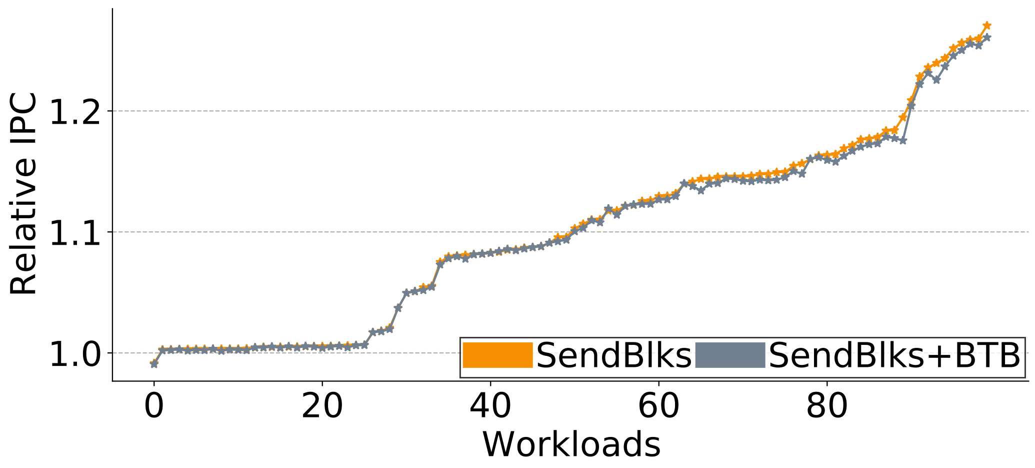


Figure 7.7: BTB+Blocks IPC

We observed that presending can effectively supply a small-sized L1 BTB with the necessary entries, resulting in performance comparable to that of an infinite BTB in many cases. Next, we evaluate the performance of presending BTB entries to a small-sized L1 BTB in conjunction with presending code cache blocks to the L1i.

In Figure 7.7, we present results demonstrating the performance improvement of Send with an 8K-entry L1 BTB when presending only code cache blocks to a 32KB L1i (SendBlks) and with presending both BTB entries to a 512-entry L1 BTB and code cache blocks to a 32KB L1i (SendBlks+BTB). IPC results are shown relative to baseline microarchitecture with an 8K-entry L1 BTB. Send with a 512-entry BTB and presending both code cache blocks and BTB entries achieves performance comparable to an 8K-entry BTB with presending of only code cache blocks.

These results demonstrate that with Send, we can effectively keep an L1i supplied with the necessary instruction blocks and a small-sized L1 BTB supplied with the necessary BTB entries for processing these blocks of instructions.

7.3.6 Summary

This chapter proposed Instruction Presending for BTBs. We described the operational enhancements to Instruction Presending to facilitate the movement and evaluated the supply of BTB entries to the L1 BTB and associated overheads.

These results suggest that with Send, a smaller-sized primary BTB is sufficient and we can achieve a very similar performance as with an infinite BTB. Send allows for BTB entries to be moved from a secondary BTB to a small-sized primary BTB in a timely manner for assisting in the correct fetching of instructions from the L1i.

Augmenting information about control independence is likely to allow FDIP to come closer to the performance of a perfect L1i. However, it would still require a large primary BTB or orthogonal BTB management to facilitate the movement of blocks. Send can achieve this with a smaller-sized primary BTB.

Chapter 8

Conclusions

In this chapter, we review the key contributions of this dissertation, summarizing the problem, the work that preceded our research, and the key findings of this dissertation. We then outline some directions for future work. Lastly, we provide some unsubstantiated opinions regarding the design of future microprocessors, the traces used, and the simulation infrastructure.

8.1 Dissertation Contributions and Summary

Instruction supply is a critical component of any high-performance processor aiming to exploit significant amounts of instruction-level parallelism. This aspect of the processor has faced challenges with the advent of new applications and modern software development practices, which begin to overwhelm reasonably sized primary on-chip structures that facilitate effective instruction fetching. Over the past two decades, considerable research has focused on preloading primary structures with the necessary information to efficiently fetch instructions. This research addresses emerging challenges and utilizes the abundance of on-chip transistors to improve processor efficiency, predominantly through a plethora of instruction prefetching techniques [29, 30, 45, 43, 44, 62, 69, 75].

Most instruction prefetching techniques focus on preloading the L1i. Techniques such as FDIP [62] use the logic within the instruction fetch unit to get ahead in the instruction stream and preload blocks of instructions into the L1i. Most other techniques [29, 30, 43, 69, 75] establish correlations with events within the instruction fetch process, such as the state of the branch predictor, return address stack, or instruction cache accesses, and use these events to trigger the preloading of cache blocks. All of these techniques are tied to the instruction fetch process. One key limitation within instruction fetch arises from BTB misses, and to mitigate this, an orthogonal BTB management [5, 39, 45, 60] scheme is employed.

This dissertation presented an alternative approach for moving blocks of instructions and information closely related to static instructions, such as BTB and iTLB entries, from their resident locations in the memory hierarchy (lower-level caches or secondary structures) to primary structures like the L1i, iTLB, and BTB. This was done in time for processing the requisite instructions and operated independently of any processor microarchitectural events or instruction fetch processes.

We observed that high-level control flow, specifically at the call graph level, was relatively easy to capture and remained mostly unchanging for the targeted applications. Further, determination and movement of blocks to be referenced do not require a generation of a precise instruction stream, and can tolerate some imprecision. Building on these insights, this dissertation proposed a technique called *Instruction Presending* to sequence high-level control flow to know where the processor is likely to be and identify and move the code cache blocks it is likely to reference. Data was presented demonstrating that this scheme near perfectly resolves upcoming high-level control flow in many cases and ensures a timely delivery of instructions, achieving performance close to that of a perfect L1i in many scenarios.

Additionally, the dissertation also proposed utilizing the same high-level sequencing to identify and move the iTLB and BTB entries the processor is likely to reference. These entries were proactively moved to the iTLB and BTB, resulting in performance that approached that of a perfect iTLB and BTB in many cases.

The thesis of this dissertation is: (1) It proposes a method that can sequence high-level control flow near perfectly in many cases; (2) It used this technique to identify code cache blocks, iTLB, and BTB entries that the processor is likely to need, moving them to the respective primary structures, resulting in improved fetching of instructions. To substantiate this thesis, we presented the following empirical and experimental evaluation:

– **High-Level Control Flow Characteristics**: A detailed analysis of the fragment-level control flow characteristics across numerous applications.

– **Instruction Presending**: The design and implementation of a technique to sequence fragment-level control flow and identify the blocks of instructions the processor is likely to reference, independent of the logic for fetching instructions.

– **Presending Code Cache Blocks**: An evaluation of the ability to accurately resolve upcoming control flow at a high-level and a detailed evaluation of the technique to move necessary code cache blocks to the L1i, compared against state-of-the-art instruction prefetching techniques.

– **Presending iTLB/BTB Entries**: Enhancements (and an evaluation) of the technique to identify the necessary iTLB and BTB entries that the processor is likely to need and move them to the iTLB and BTB in a timely manner.

8.2 Directions for Future Work

8.2.1 Movement of Branch Predictor Entries

While we addressed key primary structures needed for efficient instruction fetching, this work did not consider branch direction predictors. Branch direction predictors are crucial for fetching instructions efficiently; however, entries in the predictor are not solely identified by the instructions. Instead, entries in the branch predictor are indexed using parts of the instruction as well as recent branch history, which makes identifying the branch predictor entries likely to be needed more involved. Moreover, the contents of the branch predictor are dynamic, unlike entries in a BTB, iTLB, or code cache blocks, which are mostly static. Branch direction predictors are also impacted by larger code footprints, necessitating large tables to store all the information required for accurate predictions. There is an opportunity for innovation in this area.

8.2.2 Duplication of Information in Program Representation

While our work presents a program representation capable of identifying the blocks of instructions, iTLB entries, and BTB entries likely to be needed by the processor, this representation does include some redundancy. For example, fragments sequenced close in time might reference the same code page, resulting in the replication of the same VPN-PTE in multiple Fragment Table entries. This duplication in our key data structures can certainly be reduced to optimize storage and efficiency.

8.2.3 Movement of Other Information for Processing

While we have focused on the information necessary to improve the efficiency of fetching instructions, the high-level sequencing we proposed can also be used to identify and move other pieces of information to enhance instruction processing. For example, this approach can be extended to move Value Prediction entries [46, 47]. By doing so, we can further optimize the performance of the processor by preloading these prediction entries. Further, the instruction blocks moved can be predecoded and analyzed to further improve processing of instructions.

8.3 Reflections

In this section, I share some observations concerning the design of future microprocessors, traces used for this work and the simulation infrastructure. These are based on our experiences and conversations with others, and are simply opinions.

8.3.1 Design of Future Microprocessors

In this dissertation, we presented a high-level sequencing technique to preload different structures in time to facilitate effective fetching of instructions. The process of instruction fetching, which creates a precise stream of instructions to execute, relies on accurately going past every control instruction in the program. Today we have wide machines capable of fetching and executing over eight instructions every cycle. These designs put more pressure on instruction fetch, which must predict the outcome of multiple control instructions every cycle. This process is inherently sequential, and there have been proposals to address this challenge [68]. Some proposals [57, 58] aim to break this sequential requirement by making the process parallel. These proposals exploit control independence in the instruction stream to enable parallel fetching of instructions.

Instruction Presending maintains high-level control information at the call graph level, allowing it to go past multiple control instructions in each step while remaining agnostic to local control. Given the high accuracy in resolving high-level control flow, exposing this information to the instruction fetch process can facilitate fetching instructions from a control-independent point following a subsequent call or return instruction. This could result in higher collective fetch bandwidth when fetching instructions from these control-independent points in the instruction stream.

Beyond instruction fetching, this also opens up the opportunity to rename [58] and execute independent instructions [73] from the control-independent portion of the instruction stream even before prior instructions have been fetched. This could further allow for moving away from monolithic backends.

Overall, exploiting control independence at the call graph level opens up opportunities to revisit many old ideas and rethink and improve the microarchitectures for tomorrow.

8.3.2 Application Traces

While the commercial traces were extremely useful for understanding and developing the technique proposed in this thesis, their anonymized nature posed challenges for achieving a more detailed understanding. For instance, identifying which program constructs lead to discontiguous block regions.

Although anonymized traces provided by industry are sufficient for developing a general-purpose solution, knowing the specific applications these traces correspond to would likely make the results more exciting.

8.3.3 Simulation Infrastructure

The use of ChampSim, a trace-driven simulation infrastructure compatible with these traces, streamlined the implementation and evaluation of the technique. It avoided the complications associated with running applications on a typical execution-driven simulation infrastructure. This approach simplified many aspects of the work. However, it is worth noting that some benefits, particularly those related to IPC performance, may be overestimated due to the lack of modeling certain features, such as wrong-path execution.

Bibliography

[1] Champsim simulator: http://github.com/Champsim/Champsim, March 2023.

[2] The second championship value prediction: https://www.microarch.org/cvp1/, nov. 2020.

[3] Zen5 microarchitecture: https://www.tomshardware.com/tech-industry/deep-dives-into-amd-zen-5-nvidia-blackwell-and-intel-lunar-lake-architectures-coming-at-hot-chips-2024, july. 2024.

[4] T. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. Shen. Hardware support for prescient instruction prefetch. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 84–84, 2004. doi: 10.1109/HPCA.2004.10028.

[5] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito. The ibm z15 high frequency mainframe branch predictor industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39. IEEE, 2020.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, number CONF, pages 266–277, 1999.

[7] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary. Transparent control independence (tci). *ACM SIGARCH Computer Architecture News*, 35(2):448–459, 2007.

[8] V. A. Alfred, S. L. Monica, and D. U. Jeffrey. *Compilers principles, techniques & tools*. pearson Education, 2007.

[9] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems (TOCS)*, 21(4):412–444, 2003.

[10] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad. Divide and conquer frontend bottleneck. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 65–78. IEEE, 2020.

[11] T. Asheim, B. Grot, and R. Kumar. A storage-effective btb organization for servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1153–1167. IEEE, 2023.

[12] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 342–351, 1992.

[13] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 643–656. IEEE, 2018.

[14] 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 *Proceedings of the 46th International Symposium on Computer Architecture*, pages 462–473, 2019.

[15] R. Bhargava and K. Troester. Amd next generation" zen 4" core and 4 th gen amd epyc™ server cpus. *IEEE Micro*, 2024.

[16] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito. Two level bulk preload branch prediction. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 71–82. IEEE, 2013.

[17] I. Burcea and A. Moshovos. Phantom-btb: a virtualized branch target buffer design. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 313–324. ACM New York, NY, USA, 2009.

[18] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–167. ACM New York, NY, USA, 2008.

[19] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. *ACM SIGARCH Computer Architecture News*, 19(3):276–286, 1991.

[20] B. Calder and D. Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408, 1994.

[21] Q. Cao, P. Trancoso, J.-L. Larriba-Pey, J. Torrellas, R. Knighten, and Y. Won. Detailed characterization of a quad pentium pro server running tpc-d. In *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, pages 108–115. IEEE, 1999.

[22] G. Chacon, N. Gober, K. Nathella, P. V. Gratz, and D. A. Jiménez. A characterization of the effects of software instruction prefetching on an aggressive front-end. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 61–70. IEEE, 2023.

[23] P.-Y. Chang, E. Hao, and Y. N. Patt. Target prediction for indirect jumps. *ACM SIGARCH Computer Architecture News*, 25(2):274–283, 1997.

[24] I.-C. K. Chen, C.-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, pages 593–601. IEEE, 1997.

[25] C.-Y. Cher and T. Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 4–15. IEEE, 2001.

[26] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. *ACM SIGARCH Computer Architecture News*, 26(3):142–153, 1998.

[27] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 129–140. IEEE, 2004.

[28] M. Evers, L. Barnes, and M. Clark. The amd next-generation "zen 3" core. *IEEE Micro*, 42(3):7–12, 2022.

[29] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 1–10. IEEE, 2008.

[30] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive instruction fetch. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 152–162, 2011.

[31] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim. The championship simulator: Architectural sim-

ulation for education and competition. *arXiv preprint arXiv:2210.14324*, 2022.

[32] S. Harizopoulos and A. Ailamaki. Steps towards cache-resident transaction processing. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 660–671, 2004.

[33] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. University of California, Berkeley, 1987.

[34] W. Hsu and J. Smith. Prefetching in supercomputer instruction caches. In *Supercomputing'92*, pages 588–597, 1992.

[35] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo. Re-establishing fetch-directed instruction prefetching: An industry perspective. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 172–182. IEEE, 2021.

[36] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206. IEEE, 2001.

[37] P. Kallurkar and S. R. Sarangi. ptask: A smart prefetching scheme for os intensive applications. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.

[38] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[39] C. Kaynak, B. Grot, and B. Falsafi. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 166–177, 2015.

[40] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 15–26, 1998.

[41] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159. IEEE, 2020.

[42] T. A. Khan, N. Brown, A. Sriraman, N. K. Soundararajan, R. Kumar, J. Devietti, S. Subramoney, G. A. Pokam, H. Litz, and B. Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 816–829, 2021.

[43] A. Kolli, A. Saidi, and T. F. Wenisch. Rdip: Return-address-stack directed instruction prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 260–271. IEEE, 2013.

[44] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 493–504. IEEE, 2017.

[45] R. Kumar, B. Grot, and V. Nagarajan. Blasting through the front-end bottleneck with shotgun. *ACM SIGPLAN Notices*, 53(2):30–42, 2018.

[46] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, pages 226–237. IEEE, 1996.

[47] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference*

on Architectural support for programming languages and operating systems, pages 138–147, 1996.

[48] J. L. Lo, L. A. Barroso, S. J. Eggers, K. Gharachorloo, H. M. Levy, and S. S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 39–50. IEEE, 1998.

[49] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 182–193. IEEE, 1998.

[50] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6): 190–200, 2005.

[51] S. McFarling. Program optimization for instruction caches. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 183–191, 1989.

[52] S. McFarling. Combining branch predictors. Technical report, Citeseer, 1993.

[53] A. Moshovos. *Memory dependence prediction*. PhD thesis, PhD thesis, University of Wisconsin-Madison, 1998.

[54] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 181–193, 1997.

[55] S. Muchnick. *Advanced compiler design implementation*. Morgan kaufmann, 1997.

[56] S. Murthy and G. S. Sohi. Instruction block movement with coupled high-level program sequencing. *arXiv preprint arXiv:2406.06738*, 2024.

[57] P. Oberoi and G. Sohi. Out-of-order instruction fetch using multiple sequencers. In *Proceedings International Conference on Parallel Processing*, pages 14–23. IEEE, 2002.

[58] P. S. Oberoi and G. S. Sohi. Parallelism in the front-end. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 230–240, 2003.

[59] G.-H. Park, O.-Y. Kwon, T.-D. Han, S.-D. Kim, and S.-B. Yang. An improved lookahead instruction prefetching. In *Proceedings High Performance Computing on the Information Superhighway. HPC Asia'97*, pages 712–715. IEEE, 1997.

[60] A. Pellegrini. Arm neoverse n2: Arm's 2 nd generation high performance infrastructure cpus and system ips. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–27. IEEE, 2021.

[61] D. N. Pnevmatikatos, M. Franklin, and G. S. Sohi. Control flow prediction for dynamic ilp processors. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 153–163. IEEE, 1993.

[62] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 16–27. IEEE, 1999.

[63] A. Ros and A. Jimborean. The entangling instruction prefetcher. *IEEE Computer Architecture Letters*, 19(2):84–87, 2020.

[64] A. Ros and A. Jimborean. A cost-effective entangling prefetcher for instructions. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 99–111. IEEE, 2021.

[65] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, et al. Intel alder lake cpu architectures. *IEEE Micro*, 42(3):13–19, 2022.

[66] J. Rupley, B. Burgess, B. Grayson, and G. D. Zuraski. Samsung m3 processor. *IEEE Micro*, 39(2):37–44, 2019.

[67] A. Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127, 2011.

[68] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *ACM SIGPLAN Notices*, 31(9):116–127, 1996.

[69] A. J. Smith. Cache memories. *ACM Computing Surveys (CSUR)*, 14(3): 473–530, 1982.

[70] J. E. Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[71] J. E. Smith and J. R. Goodman. Instruction cache replacement policies and organizations. *IEEE Transactions on Computers*, 34(03):234–241, 1985.

[72] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, 1995.

[73] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, 1995.

[74] L. Spracklen, Y. Chou, and S. G. Abraham. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *11th International Symposium on High-Performance Computer Architecture*, pages 225–236. IEEE, 2005.

[75] V. Srinivasan, E. S. Davidson, G. S. Tyson, M. J. Charney, and T. R. Puzak. Branch history guided instruction prefetching. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 291–300. IEEE, 2001.

[76] A. Sriraman, A. Dhanotia, and T. F. Wenisch. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

[77] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. *ACM SIGPLAN Notices*, 35(11):257–268, 2000.

[78] D. Tarjan and K. Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM transactions on architecture and code optimization (TACO)*, 2(3):280–300, 2005.

[79] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. *ACM SIGARCH Computer Architecture News*, 23(2):345–356, 1995.

[80] G. Vavouliotis, L. Alvarez, B. Grot, D. Jiménez, and M. Casas. Morrigan: A composite instruction tlb prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1138–1153, 2021.

[81] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[82] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[83] C. Xia and J. Torrellas. Instruction prefetching of systems codes with layout optimized for reduced cache misses. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 271–282, 1996.

[84] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, 1991.

[85] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *ACM SIGMICRO Newsletter*, 23(1-2):129–139, 1992.

[86] Y. Zhang, S. Haga, and R. Barua. Execution history guided instruction prefetching. In *Proceedings of the 16th international conference on Supercomputing*, pages 199–208, 2002.

[87] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 2–13, 2001.