Criticality-Aware Front-end

Bhargav Reddy Godala

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Professor David I. August

September 2024

Abstract

Code footprints continue to grow faster than instruction caches, putting additional pressure on existing front-end structures. Even with aggressive front-ends with fetch-directed instruction prefetching (FDIP), modern processors experience significant front-end stalls. Due to the end of Moore's Law, increasing cache sizes raises critical path latency, with modest returns for scaling instruction cache sizes. This dissertation aims to address front-end bottlenecks by making two key observations. In FDIP-enabled processors, cache misses have unequal costs, and a small fraction of critical instruction cache lines contribute to most of the front-end stalls.

EMISSARY, the pioneering cost-aware replacement policy tailored for the L1 Instruction Cache (L1I), defies conventional wisdom by presenting a groundbreaking approach. Unlike traditional replacements, EMISSARY demonstrates performance enhancements even amidst increased instruction cache misses. However, EMISSARY proves to be less effective when applied to datacenter workloads characterized by large code footprints. This is due to datacenter workloads having more critical lines greater than the capacity of L1I. This dissertation first presents improved EMISSARY-L2, the first criticality-aware cache replacement family of policies specifically designed for datacenter workloads. Observing that modern architectures entirely tolerate many instruction cache misses, EMISSARY-L2 resists evicting those cache lines whose misses cause costly decode starvations from L2. In the context of a modern FDIP-enabled processor, EMISSARY-L2 delivers an impressive 3.24% geomean speedup (up to 23.7%) and a geomean energy savings of 2.1% (up to 17.7%) when evaluated on datacenter workloads. This speedup is 21.6% of the speedup obtained by an unrealizable L2 cache with a zero-cycle miss latency for all capacity and conflict instruction misses.

This dissertation then proposes Priority Directed Instruction Prefetching (PDIP), a novel cost-ware instruction prefetching technique that complements FDIP by issuing prefetches for targets along the resteer path where FDIP stalls occur. PDIP identifies these targets and associates them with a trigger for future prefetch. When paired with EMISSARY-L2, PDIP achieves a geomean IPC speedup of 3.7% across a set of datacenter workloads using a budget of only 43.5KB. PDIP achieves 62% of the ideal prefetching performance.

Acknowledgements

I extend my sincerest gratitude to my advisor, Prof. David I. August, whose unwavering support and unique problem-solving approach have been instrumental throughout this journey. His optimism in tackling even the most challenging problems has been truly inspiring, emphasizing the importance of addressing issues at their core rather than focusing solely on surface-level solutions.

I am deeply thankful for the guidance and mentorship provided by Gilles A. Pokam. His dedication to improving the quality of my work and providing timely feedback have been invaluable. Gilles has not only supported me in my research but has also played a significant role in expanding my professional network, introducing me to experts both within Intel and academia.

A heartfelt thanks to my dissertation committee, including Prof. Margaret Martonosi, Prof. David Wentzlaff, Gilles A. Pokam, and Svilen Kanev, for their valuable insights and contributions to my research. Special gratitude goes to Prof. Simone Campanoni for his ongoing support and patience throughout our collaboration.

I am indebted to Svilen Kanev for his invaluable feedback and guidance, which have helped me navigate through challenging phases of my research. Similarly, Jared Stark's contributions to improving the modeling of the CPU front-end have been foundational to the progress of my thesis.

Prof. Dean Tullsen's assistance in refining the quality of my writing and research outcomes for the PDIP paper is deeply appreciated. I am also grateful to Sankara for his dedication in seeing the PDIP project to completion.

I extend my thanks to Prof. Andre Seznec, Prof. Paul Gratz, Prof. Daniel Jimenez, and Prof. Ashish Venkat for their collaboration on various projects. Additionally, I am thankful to Mike Chu and Hassan Muhammad from AMD Research for their support and guidance during my internship.

I would like to acknowledge the members of the Liberty Research Group, particularly Sotiris, Nayana, Hansen, Greg, and Ziyang, for their camaraderie and support during my time at Princeton. Special thanks to Susan, Yebin, Ishita, Yucan, and Sophia for their enthusiasm and energy.

I am grateful to the administrative staff of the Department of Computer Science at Princeton, especially Nicki Mahler, for their assistance with bureaucratic matters and timely reminders. I also extend my thanks to the Davis International Center staff for their support with immigration-related issues.

I express deep gratitude towards my parents for their unwavering love and support, for the sacrifices they made to ensure my education and opportunities, which paved the way for my journey, and for the values they imparted to me. Additionally, I extend my appreciation to my brother for his constant belief in me.

I gratefully acknowledge the generous financial support for my graduate school work by a variety of sources: the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract numbers DE-SC0022138 and DE-SC0022268; the National Science Foundation under Grant numbers CCF-2107257, CCF-2118708, CCF-2107042, and CCF-1908488; Intel; and the National Science Foundation under Grant CCF-2107257.

To my parents and my brother

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the contemporary era marked by the widespread adoption of cloud computing and the unprecedented explosion of data, the demand for computational resources has reached unprecedented levels and is only increasing. At the forefront of addressing this insatiable appetite for compute power are data centers, which play a pivotal role in meeting the escalating requirements of a digital age. However, the conventional trajectory of technological progress faces a formidable challenge with the end of Moore's Law, making it increasingly difficult to keep pace with the surging demand for computational capabilities.

In the face of this challenge, even marginal performance gains in data center processors carry substantial implications. Each incremental improvement holds the potential to yield significant savings in operational costs, making the pursuit of enhanced processing efficiency a strategic imperative.

The code footprints of the datacenter applications have been growing at $\approx$30% per year [44] whereas the instruction cache sizes are not growing at the same rate. This puts enormous pressure on the CPU front-end, which leads to poor performance. A significant number of issue slots are wasted due to front-end bottlenecks. This is an ever-growing problem due to growing instruction footprints. This problem is challenging to address as traditional scaling-based solutions are no longer feasible due to the end of Moore's Law scaling, which only yields diminishing returns. This dissertation studies the criticality aspect of instruction cache lines to drive the performance of datacenter workloads.

1.1 Increase in the Front-end Pressure due to Growing Instruction Footprints

Modern processors implement a decoupled front-end, aka fetch-directed instruction prefetching (FDIP) [66], as an attempt to reduce front-end bottlenecks [62, 37, 72, 20]. With FDIP, the L1I fill is disassociated from its demand access, thus allowing the front-end to aggressively prefetch along the predicted path with very little sensitivity to decode and backend back pressure. This allows the FDIP prefetcher to hide most or all of the latency of L1I misses. Despite FDIP in modern processors several server applications spend significant time in the front-end. Figure 1.1 shows top down [78] analysis of several server workloads. It shows that out of all issue slots, only 12% are used in retiring useful instruction. A staggering 48% (≈50%) of issue, slots are wasted due to front-end, and 32% of issue slots are lost due to backend. This shows that front-end is still a big problem due to the large code footprints of modern datacenter workloads. Since instruction footprints are growing at much higher rates than the size of caches, this issue is becoming more pronounced over time.

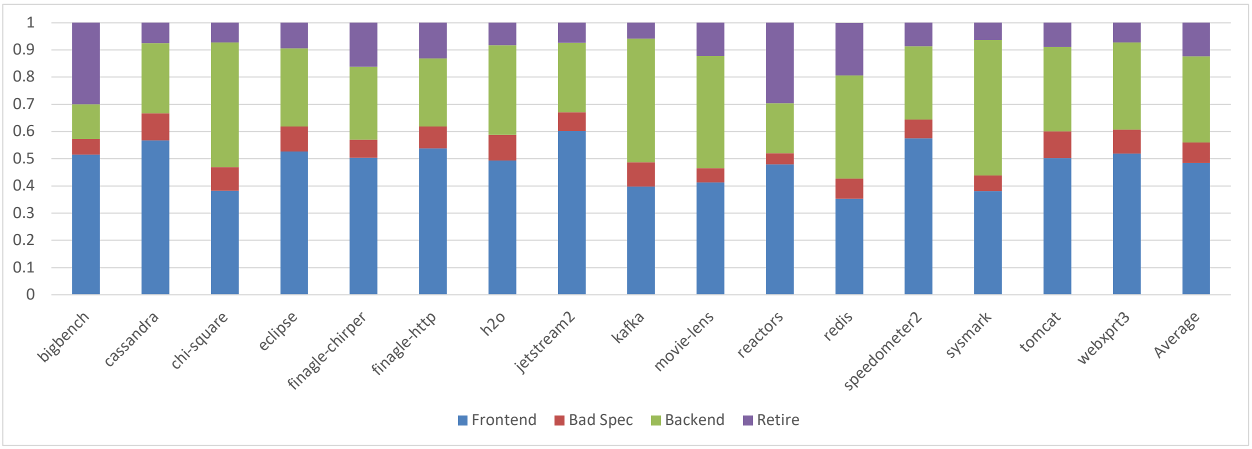


Figure 1.1: Top-Down analysis of several server workloads

1.2 End of Moore's Law: Need for Algorithm-based Solution instead of Scaling-based Solutions

Moore's Law states that transistors in a given area doubles about every two years. In Moore's Law era, the size of the cache could be doubled at the same access latency. Due to the end of Moore's Law, increasing cache sizes comes at the cost of increased latency. The Instruction Cache sits in the critical path of the front-end. An increase in the latency of the Instruction Cache would lead to lower performance. Figure 1.2 shows performance gain of datacenter workloads when the Instruction Cache is scaled, and the access latency is the same as the baseline 32KB Instruction Cache. Interestingly, the results show that scaling the Instruction Cache by sixteen times only yields a performance gain of 4.5%, which is significantly lower than expected given the size of the cache. This suggests that traditional scaling-based solutions are not effectively addressing the significant front-end bottlenecks faced by datacenter workloads.

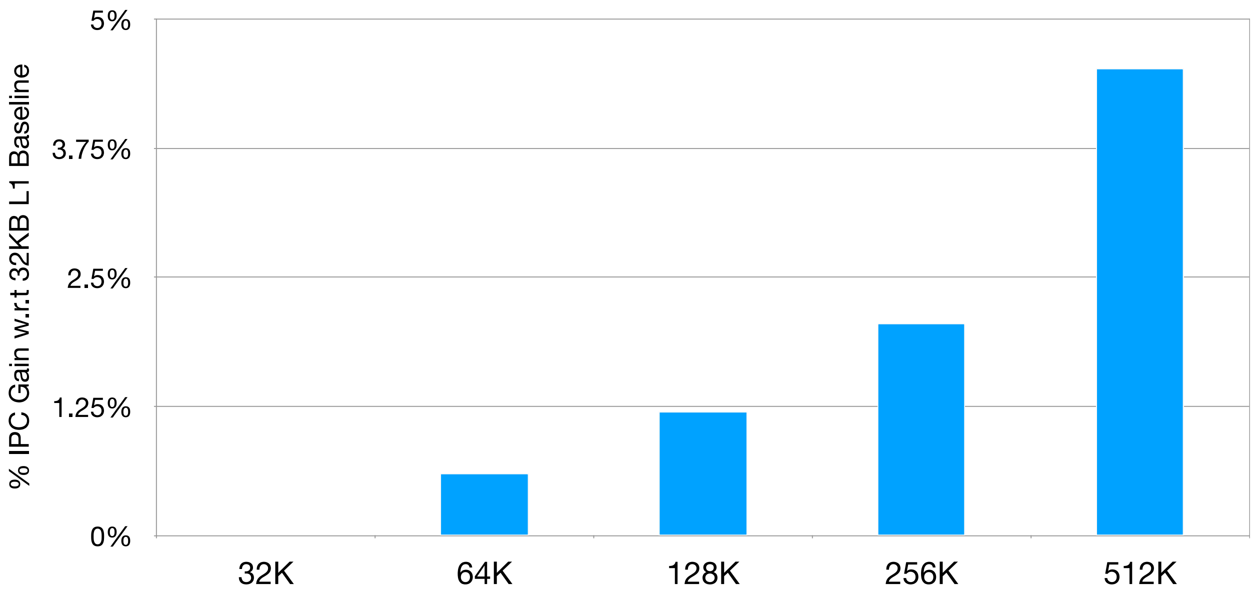


Figure 1.2: Speedup in % IPC gain w.r.t 32K Instruction Cache system of several server workloads.

In light of these challenges, it becomes increasingly essential to explore algorithm-based solutions to improve performance in the post-Moore's Law era. These solutions may involve optimizing instruction fetching algorithms, improving prefetching strategies, or implementing more efficient instruction cache management techniques to mitigate front-end bottlenecks and enhance overall performance.

1.3 Front-end Bottlenecks in Large Code Footprint Workloads

As scaling-based solutions are showing diminishing returns, a deeper understanding of front-end bottlenecks becomes imperative to identify the root cause. Figure 1.3 provides a top-down analysis of a well-known database application Cassandra [1], a datacenter workload, revealing significant front-end limitations. On the left side of Figure 1.3, it is evident that more than half of the issue slots are wasted due to front-end bound events.

The right side of Figure 1.3 offers a further breakdown of various front-end events. It highlights that branch resteers and Instruction Cache misses contribute to 33% and 31% of lost issue slots, respectively. Together, they account for 64% of all front-bound events. Instruction Cache size emerges as one of the major bottlenecks. However, simply increasing cache size does not significantly improve performance. While a larger cache could help reduce cache misses, it would not necessarily address the cache misses that are critical for performance improvement. In an aggressive FDIP pipeline, the latency of an instruction cache miss can be fully tolerated without causing front-end stalls. Consequently, a conventional replacement policy focused on reducing misses may retain cache lines with minimal impact on performance when provided with a larger capacity. In the presence of an aggressive FDIP front-end, it becomes crucial to allocate limited front-end resources to cache lines critical for performance.

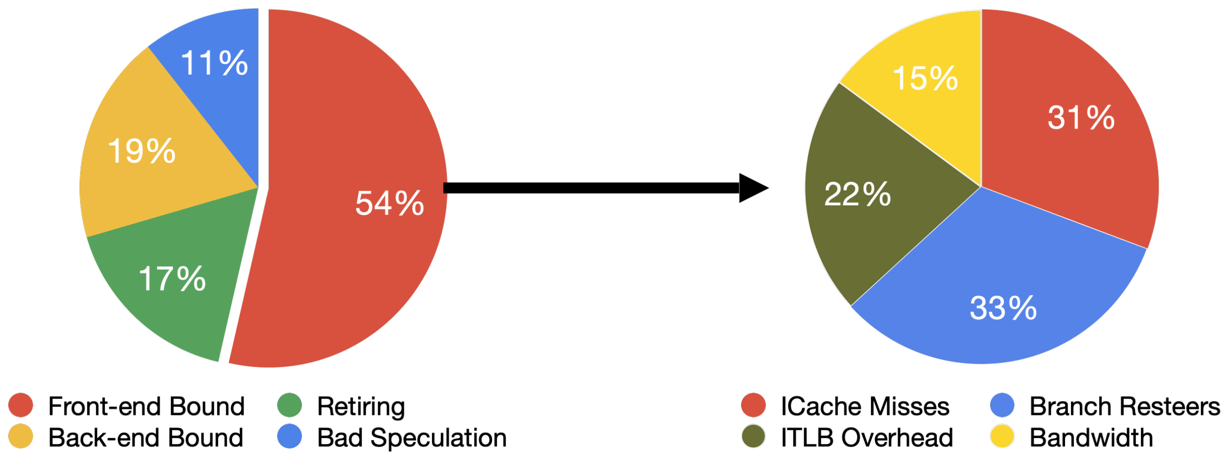


Figure 1.3: Top-Down analysis of Cassandra benchmark

1.4 Dissertation Contributions

This dissertation aims to address the growing instruction footprint problem using an algorithm-based approach to improve the performance of the CPU front-end of the datacenter applications. This dissertation studies the root cause of the front-end bottlenecks in datacenter workloads. The first step in studying front-end bottlenecks is to model state-of-the-art front-end (FDIP). The FDIP model is implemented in the gem5 simulator. Which is the first execution-driven model of the decoupled front-end. Observing that only a small fraction of critical cache lines are responsible for the majority of stalls in the CPU. This dissertation also proposes a novel criticality-aware replacement policy called EMISSARY-L2, the first such policy designed for datacenter workloads and a novel criticality-aware instruction prefetcher called PDIP, which complements FDIP.

1.4.1 State-of-the-art Front-end

The Fetch Directed Instruction Prefetching [67] technique was introduced in 1999 but no execution-driven simulators for superscalar processors have been implemented. Ishii et al. [41] shows that instruction prefetchers used in the first instruction prefetching championship do not show any significant performance gains when used with an FDIP-enabled processor model. This shows that front-end studies need to use the FDIP model.

Prior works used the FDIP model in a trace-driven simulator, which ignores instructions executed in the mispredicted path. Trace-driven simulation models often lack accuracy in capturing this phenomenon due to the trade-off between speed and accuracy, leading to oversimplified representations of the CPU behavior. Previous research [70, 31] has typically focused on either the positive or negative consequences of wrong paths, but not both simultaneously. This dissertation introduces a novel approach to modeling the wrong path in the trace-driven simulator. Detailed design is described in Chapter 5. Using this model, we empirically show that FDIP with the wrong path model shows a 32.8% increase in the Instruction Cache Misses Per Kilo Instructions (MPKI) compared to the model that ignores the wrong path. This underscores the necessity for front-end optimizations to account for FDIP with wrong path modeling to achieve accurate simulation results. Thus, our work implemented FDIP in an execution-driven Out-of-Order CPU of a widely used gem5 [27] simulator. Detailed design with several novel optimizations is described in Chapter 2. We show that FDIP design shows up to 30% speedup compared to baseline without FDIP. This design helps in realizing a key observation that ”not all misses are equal.”

1.4.2 Criticality-Aware Replacement Policy for Datacenter Wrokloads

For decades, architects have designed cache replacement policies to reduce cache misses. Since not all cache misses affect processor performance equally, researchers have also proposed cache replacement policies focused on reducing the total miss cost rather than the total miss count. Prior cost-aware replacement policy EMISSARY [57] works at L1 Instruction Cache, which is ineffective for datacenter workloads with large code footprints. This is due to datacenter workloads have a higher volume of critical lines, and long reuse distances cannot be preserved in the Instruction Cache before they can be used. This dissertation presents an improved EMISSARY-L2 , the first criticality-aware cache replacement family of policies specifically designed for datacenter workloads. Observing that modern architectures entirely tolerate many Instruction Cache misses, EMISSARY-L2 resists evicting those cache lines whose misses cause costly decode starvations when the backend is idle. In the context of a modern processor with fetch-directed instruction prefetching and other aggressive front-end features, EMISSARY-L2 delivers an impressive 3.24% geomean speedup (up to 23.7%) and a geomean energy savings of 2.1% (up to 17.7%) when evaluated on widely used server applications with large code footprints. This speedup is 21.6% of the total speedup obtained by an unrealizable L2 cache with a zero-cycle miss latency for all capacity and conflict instruction misses.

1.4.3 Criticality-Aware Instruction Prefetcher

Modern server workloads have large code footprints that are prone to front-end bottlenecks due to instruction cache capacity misses. Even with the aggressive fetch-directed instruction prefetching (FDIP) implemented in modern processors, there are still significant front-end stalls due to I-Cache misses. A major portion of misses that occur on a BPU-predicted path are tolerated by FDIP without causing stalls. Prior work on instruction prefetching, however, has not been designed to work with FDIP processors. Their singular goal is to reduce I-Cache misses, whereas FDIP processors are designed to tolerate them. Designing an instruction prefetcher that works in conjunction with FDIP requires identifying the fraction of cache misses that impact front-end performance (that are not fully hidden by FDIP) and only targeting them.

In this dissertation, we propose a Priority Directed Instruction Prefetching (PDIP), a novel criticality-aware instruction prefetching technique that complements FDIP by issuing prefetches for only targets where FDIP struggles – along the resteer path of front-end stall-causing events. PDIP identifies these targets and associates them with a trigger for future prefetch. At a 43.5KB budget, PDIP achieves up to 5.1% IPC speedup on important workloads such as `cassandra` and a geomean IPC speedup of 3.2% across 16 benchmarks.

1.5 Published Material

Criticality-Aware Cache Replacement Policy for datacenter workloads is the improvement over prior work EMISSARY [57]. The improved version is co-authored with Nayana is published in [58]. Published version [58] contains results and analysis obtained on datacenter workloads, which are new contributions. Chapter 3 describes an improved technique called EMISSARY-L2 by referring to the original technique as EMISSARY-L1. Chapter 3 borrows content from [58], which is either shown in footnotes or at the beginning of the section. Criticality-Aware Instruction Prefetching (Chapter 4) has been published in [36].

Chapter 2

Current State-of-the-art Product Front-end

Modern processors employ decoupled front-end design where the The Branch Predictor Unit (BPU) is decoupled from the Instruction Fetch Unit (IFU) which allows the front-end to prefetch instructions along the predicted path. The decoupled front-end is also known as Fetch Directed Instruction Prefetching (FDIP). FDIP is a structural change to the front-end pipeline of the processor and has been a key feature in the industry for over a decade [40, 41, 62]. Thus, FDIP is the stat-of-the-art front-end that should be used as a baseline for any front-end related work.

A key contribution and distinguisher of this dissertation is the fact that we faithfully model a very aggressive processor front-end and used it as a baseline, extending gem5's O3CPU model to implement Fetch Directed Instruction Prefetching (FDIP), thus supporting a decoupled front-end. A detailed design of FDIP with several novel optimizations is presented in this chapter. As gem5 is execution-driven, the wrong path effects of such resteers are also accurately modeled. An aggressive front-end can cause increased pollution of instruction when a predicted path is wrong. The effect of wrong path pollution in the presence of FDIP is studied using a trace-driven ChampSim simulator.

An important component of a decoupled front-end is the depth of decoupling, which determines how many cache lines are prefetched in the predicted path. A sufficiently deep decoupled front-end can hide the latency of an Instruction Cache (I-Cache) miss. Thus, in the presence of an aggressive decoupled front-end, a small subset of I-Cache misses expose front-end latency, which is critical for improving performance. This chapter concludes with a discussion on criticality, which is used in the rest of the dissertation.

2.1 Fetch Directed Instruction Prefetching (FDIP)

A traditional CPU front-end features a coupled design where the Instruction The Fetch Unit (IFU) is coupled with the Branch Predictor Unit (BPU). When the fetch engine encounters a branch instruction, it queries BPU to know whether the branch is taken. If the branch is taken, then BPU also provides the target of the branch so that the fetch engine can fetch those cache lines. The state-of-the-art front-end employs a branch predictor decoupled design. This design helps prefetch cache lines in the predicted path.

As the branch predictors keep getting better, it is intuitive to prefetch lines along the predicted path. The BPU keeps going ahead even when the fetch is stalling. This can be efficiently achieved by decoupling BPU from IFU. However, decoupling BPU from IFU requires several changes to the front-end stages. The BPU sends its prediction outcomes through the Fetch Target Queue to Fetch (FTQ). Fetch stage needs to be modified to operate on Fetch Target Queue entries. A Prefetch Engine reads entries from FTQ, and issues prefetch requests. One key advantage of this design is that the prefetch engine can continue issuing prefetch requests even when the fetch is stalling. Thus, FTQ helps in hiding latencies of later entries in the shadow of the prior miss. A sufficiently large FTQ can hide I-Cache miss latency even when there are no prior misses when it occurs at the tail end of the FTQ.

Figure 2.1 shows a generic traditional front-end pipeline. The BPU is within the IFU. Fetch Engine accesses the I-Cache when a new line is needed. Figure 2.2 shows a generic

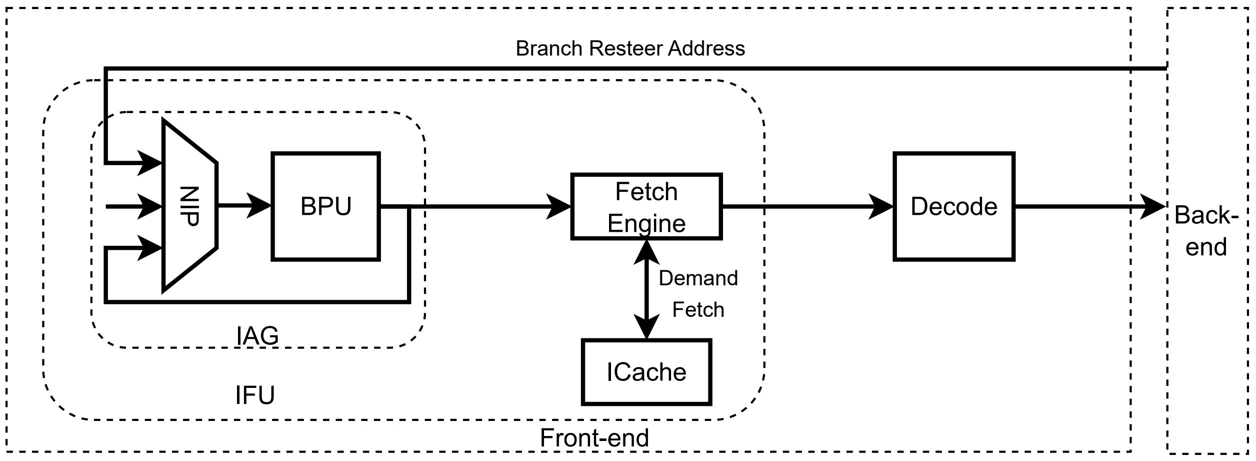


Figure 2.1: Generic traditional front-end pipeline

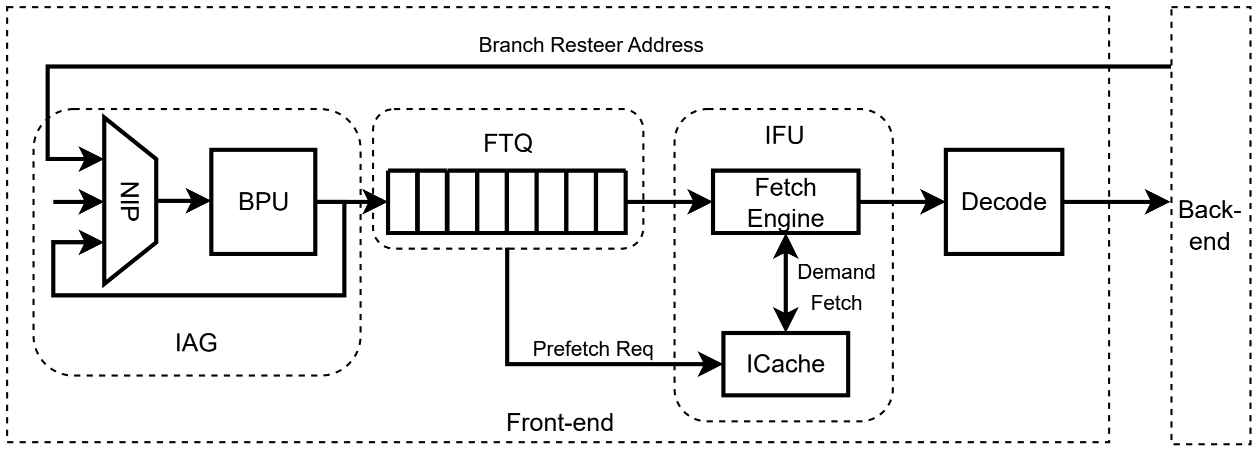


Figure 2.2: Generic FDIP front-end pipeline

front-end with an FDIP pipeline. BPU is decoupled from IFU, an FTQ is inserted between the Instruction Address Generation (IAG) unit and IFU.

Operations performed by various key components used to enable FDIP are described in detail in the following sections. Several optimizations were implemented to improve further the performance of FDIP, which are discussed in Section 2.2.

2.1.1 Fetch Target Queue

The Fetch Target Queue (FTQ) serves as a pivotal component, enabling the effective decoupling of the Branch Prediction Unit (BPU) and the Fetch Unit. The depth of the FTQ plays a crucial role in determining the extent of this decoupling. Prior works [40] have determined that an FTQ depth of 24 entries gives optimal performance. We have used 24-entry depth FTQ in all experiments presented in this dissertation. En-queuing an entry in FTQ is a serial process. Each entry of FTQ contains a begin address, branch address, and target of the branch. Each entry starts the target of the previous branch and terminates at the new branch, which BPU has predicted taken. The FTQ resembles a linked list data structure. Each entry of FTQ represents a basic block. Each element's taken target is the begin address of the next entry. Figure 2.3 shows three entries of FTQ as implemented in gem5. Note that a new element can be inserted only when the target of the current branch is known. Thus, en-queuing FTQ is a serial process. Each entry could span one or more I-Cache lines. A prefetch engine would prefetch cache lines reading FTQ entries, and a new fetch unit is responsible for de-queuing FTQ.

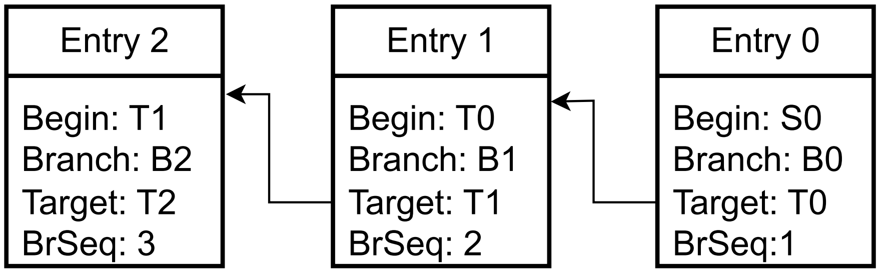


Figure 2.3: FTQ Entries

2.1.2 Prefetch Engine

The prefetch engine reads entries from FTQ, and issues prefetch requests. It is the key component in realizing the performance benefit of a decoupled front-end. Since the FTQ entry spans more than one cache line, the FTQ entry needs to be converted to its respective cache lines before a prefetch request can be issued. The cache lines created using an FTQ entry are inserted into another structure called Prefetch Buffer. Prefetch Engine issues prefetch requests corresponding to the entries in Prefetch Buffer each cycle. The I-Cache is probed before issuing each prefetch request, Thereby avoiding prefetching entries that are already present. Fetch Buffer is used to track cache lines for which prefetch requests are issued.

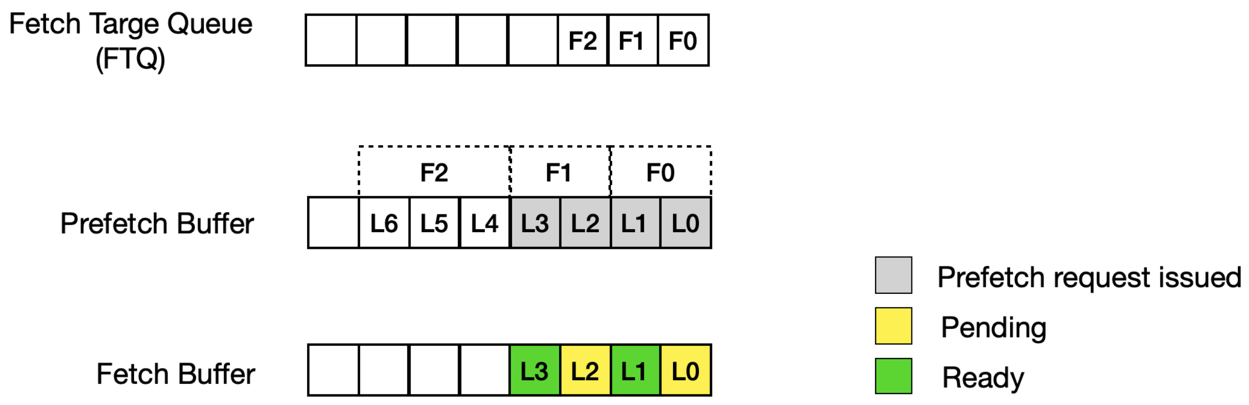


Figure 2.4: Snapshot of a Prefetch Engine State with FTQ Entries and Prefetch Buffer.

Figure 2.4 shows a sample prefetch engine state. The FTQ contains three entries $F0$, $F1$, $F2$. The Prefetch Buffer contains cache lines corresponding to entries in FTQ. The cache lines $L0$ and $L1$ correspond to FTQ head entry $F0$. The Fetch Buffer shows that prefetch requests for cache lines $L0$, $L1$, $L2$, and $L3$ are issued. Cache lines $L1$ and $L3$ are ready, and $L0$ and $L2$ are pending.

2.1.3 New Fetch Stage

The enhanced fetch stage now operates on entries within the Fetch Target Queue (FTQ) rather than cache lines. As part of its responsibilities, the fetch unit is tasked with dequeuing the head of the FTQ once the corresponding element has been processed. Additionally, the fetch engine is equipped to flush the FTQ in the event of a branch misprediction in the pipeline. Algorithm 1 outlines the modifications made to the fetch processing algorithm to incorporate the FTQ. In Line 1, the algorithm checks whether the current Program Counter (PC) falls within the range of the head element in the FTQ. If the PC is outside this range, indicating a mismatch in the FTQ state, both the FTQ and branch predictor states are flushed and corrected accordingly. Conversely, if the PC is within the FTQ head's range, the algorithm verifies whether the current PC matches the branch address of the FTQ head. A match signifies the completion of processing for the FTQ entry; hence, the entry is dequeued, and the respective pointers are updated. The fetch engine seamlessly transitions to processing a new element in the subsequent cycle. This integration of the FTQ enhances the fetch processing algorithm, providing a a more versatile and efficient mechanism for handling branch predictions and fetch operations.

Algorithm 1 Modified Fetch Engine

```
1:  if FTQ.empty() then
2:      PC += instruction.size()
3:  else
4:      if PC in FTQ.head().range() then
5:          if PC = FTQ.head().branchPC then
6:              PC ← FTQ.head().target()
7:              FTQ.pop()
8:          else
9:              PC += instruction.size()
10:         end if
11:     else
12:         FTQ.flush()
13:     end if
14: end if
```

2.1.4 Branch Target Buffer Design

The Branch Target Buffer (BTB) plays a crucial role in predicting whether a given address corresponds to a branch instruction, along with providing information about the predicted branch's target address and type. Branches can be categorized as direct or indirect, with direct branches having their targets encoded as offsets in the instruction, while indirect branches retrieve their targets from register operands or the program stack. Direct branches further subdivide into conditional and unconditional types, where unconditional branches are always taken, and conditional branches are taken based on evaluated conditions.

As an integral component of the Branch Prediction Unit (BPU), the organization of the BTB significantly impacts BPU performance. A BTB miss triggers a pipeline flush and resteer operation, incurring substantial energy and performance costs. The BTB undergoes a lookup for every Program Counter (PC) to ascertain if it corresponds to a branch.

In a decoupled front-end design, a new Fetch Queue (FTQ) entry is generated each cycle. Given a PC, the BTB is queried for all subsequent addresses until a branch is encountered. This process is resource-intensive due to the multitude of lookups required each cycle.

Consider a chain of basic blocks depicted in Figure 2.5, where each block terminates at a taken target, leading to another basic block. Figure 2.6 illustrates two BTB organization variants for this chain. In the PC-based BTB design (left), targets are inserted at the index of the branch PC, while in the Basic Block Based (BBL) BTB design (right), branch targets are inserted at the beginning address of each basic block. For instance, the basic block $BB1$ contains branch $br1$ with target $target1$ (beginning address of $BB2$). In the PC-based design, $target1$ is inserted at index $br1$. In the BBL-based design, $target2$ is inserted at index $target1$ (beginning address of $BB2$), and similarly, $target3$ at index $target2$. As the FTQ requires the branch's address, the BBL-based BTB also stores the PC of the branch. The primary advantage of the BBL-based BTB is that it requires only one lookup per cycle. Moreover, if the same branch is reached from different entry points, it is stored as distinct entries in the table.

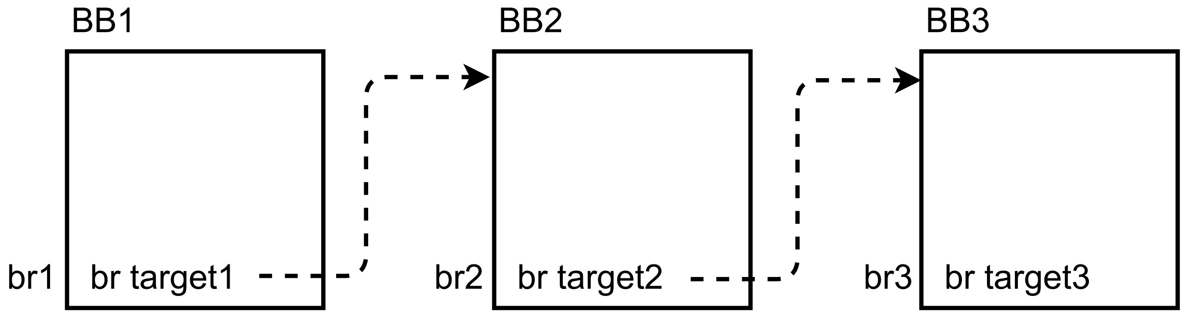


Figure 2.5: A chain of basic blocks showing branches and their targets

Index	Traget
br1	target1
br2	target2
br3	Target3

PC based BTB

Index	Target	Branch
target1	target2	br2
target2	target3	br3

BBL based BTB

Figure 2.6: BTB Organizations; PC Based BTB on the left and BBL Based design on the right

2.2 Optimizations

Implementing the decoupled pipeline incurs the cost of making the processor pipeline even deeper. In cases of incorrect branch prediction, entries prefetched along the mispredicted path may lead to cache pollution. These prefetched entries may either remain unused later in the program or displace useful lines from the I-Cache. Branch resolution can occur at various stages within the pipeline. The decode stage corrects unconditional branches and directly jumps where the target is known. On the other hand, the execute stage corrects branches affected by code execution, altering either their direction or target.

The time taken to issue the correction or resteering signal varies depending on where the signal is generated, either from the decode or execute stage. As the time to resolve a branch increases, the potential for cache pollution also rises. Enhancing the branch predictor's accuracy is crucial to improving overall front-end performance. Consequently, additional optimizations are necessary to maximize the performance potential of state-of-the-art pipelines like FDIP. We propose two novel optimizations to reduce the overall cost of misprediction redirection. To address this, we propose two innovative optimizations aimed at reducing the overall cost of misprediction redirection.

The first optimization is to introduce early prefetch correction, which identifies mispredictions in the fetch stage. This allows for prompt detection and correction of mispredictions, minimizing their impact on subsequent pipeline stages.

The second optimization is that we implement bogus invalidation of BTB entries when mispredictions are identified at the fetch stage. By invalidating these entries early in the pipeline, we prevent unnecessary lookups and fetches based on incorrect predictions, thus reducing wasted computational resources and mitigating cache pollution.

2.2.1 Early Prefetch Corrections

In the fetch stage, bytes corresponding to instructions are readily available. These bytes can be examined to detect any branches that have missed in the BTB. Specifically, only direct unconditional branches are considered, as they unconditionally alter the control flow. Conditional branches, on the other hand, necessitate invoking a conditional branch predictor to determine whether the branch is taken or not. Once an unconditional branch is identified, the Branch Prediction Unit (BPU) is corrected in the appropriate direction. This optimization strategy is termed Early Prefetch Correction.

However, the feasibility of this optimization depends on the ability to determine instruction lengths. In architectures with variable-length instruction encoding, such as x86, instruction lengths are determined in the decode stage. Therefore, this optimization cannot be executed in such architectures. Conversely, in architectures like ARM, where instruction lengths are fixed, this optimization can be effectively implemented.

2.2.2 Invalidating Bogus BTB Entries

In the x86 architecture, the Early Prefetch Correction optimization cannot be executed due to the unavailability of instruction lengths during the fetch stage. However, in the FDIP design, the Fetch Queue (FTQ) entry encompasses information about the start and end of a basic block, where the end address corresponds to a branch's Program Counter (PC). Leveraging this knowledge, the branch PC address can be utilized to verify the opcode. If the actual opcode does not correspond to a branch opcode, it indicates a bogus branch within the FTQ.

Identifying and invalidating these bogus branches from the Branch Target Buffer (BTB) can mitigate potential issues, particularly when such bogus branches occur within the loop body. This optimization ensures the accuracy of branch prediction and helps improve performance, particularly in scenarios where incorrect branch predictions could adversely affect program execution, such as within loop structures.

2.3 Impact of Wrong Path in a machine with FDIP

The core concept of FDIP is to prefetch instructions along the predicted path. These prefetched instructions can land either on the correct path or on the wrong path (mispredicted path). Prior work [31] shows that when the effects of the wrong path are ignored, the performance projection errors could be as high as 22%. An aggressive FDIP front-end can further impact performance projection errors. In the current landscape of front-end works where performance gains are less than 5%, which falls well within the error range of ignoring the wrong path, there is a need to investigate the impact of wrong.

In order to study the effects of the wrong path in the presence of a modern FDIP front-end, we use a ChampSim trace-driven simulator. ChampSim features FDIP front-end but lacks wrong path support. We modified ChampSim to model wrong path instructions and generated traces using an execution-driven gem5 simulator. Detailed modifications are described in the following section.

2.3.1 Wrong Path Model in ChampSim

ChampSim was originally designed to model a stream of instructions using the correct path. When a branch instruction is observed, the branch prediction unit (BPU) is queried regarding the branch direction and target. Since the trace contains the correct target of the branch, it is checked against the predicted target to check for misprediction. The front-end is stalled till the branch is resolved. Once the branch is resolved a constant penalty is paid to account for pipeline repair costs and then fetch is resumed again to process the stream of instructions in the correct path. Since wrong-path instructions are not seen in the pipeline; there is no need to squash or repair any structures in the pipeline.

In order to model instructions in the wrong path, all stages in the pipeline need to be modified. The fetch stage is modified to continue streaming instructions from the trace after a misprediction. The fetch stops streaming from the trace once the instructions from the correct path are observed. The fetch stage can continue after the signal to resteer is received. This signal could come either from the decode for direct unconditional branches or the execute stage for any other mispredictions.

In the decode stage, unconditional direct branches that are mispredicted can be identified. Once the mispredicting branch is found, all newer (younger) instructions following the mispredicted branch need to be squashed. The resteer signal to the fetch stage is sent, and the FTQ, decode buffer, and fetch buffer are flushed.

The execute stage handles any other mispredictions. At the execute stage, instructions are executed out of order from the Reorder Buffer (ROB). The ROB contains instructions in the program order (in-order). Instructions in the ROB following a mis-speculating instruction are on the wrong path. Once the miss-speculation is resolved, all following instructions in the ROB are flushed. All pipeline structures leading to the execute stage are flushed. Instructions in the wrong path are renamed, so the rename maps are repaired to remove stale dependencies introduced by the wrong path instructions.

Mispredictions observed on the wrong path are ignored to keep the design simple. However, wrong-path traces include corrected mispredictions in the wrong path as they are collected from the execution of another execute-driven simulator. Any effect of these corrections is already captured by wrong path traces. Thus, they are approximated using a single correction in ChampSim.

2.3.2 Impact of Wrong Path

Instructions executed in the wrong path can indeed pollute both data and instruction caches, potentially leading to pollution in higher-level caches as well. The degree of pollution is influenced by the number of instructions executed in the wrong path, which is constrained by the size of the ReOrder Buffer (ROB). This pollution is exacerbated by the aggressive nature of the FDIP pipeline, where prefetches for cache lines in the Fetch Target Queue (FTQ) could be serviced for wrong path entries while the head of the FTQ is still waiting.

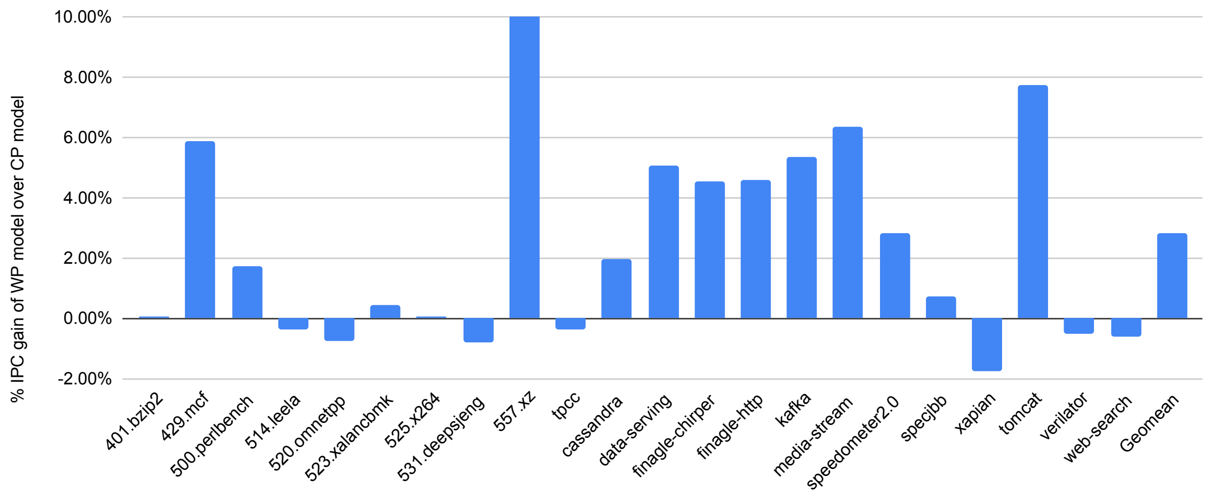


Figure 2.7: % IPC gain of various workloads with Wrong Path model over Correct Path model

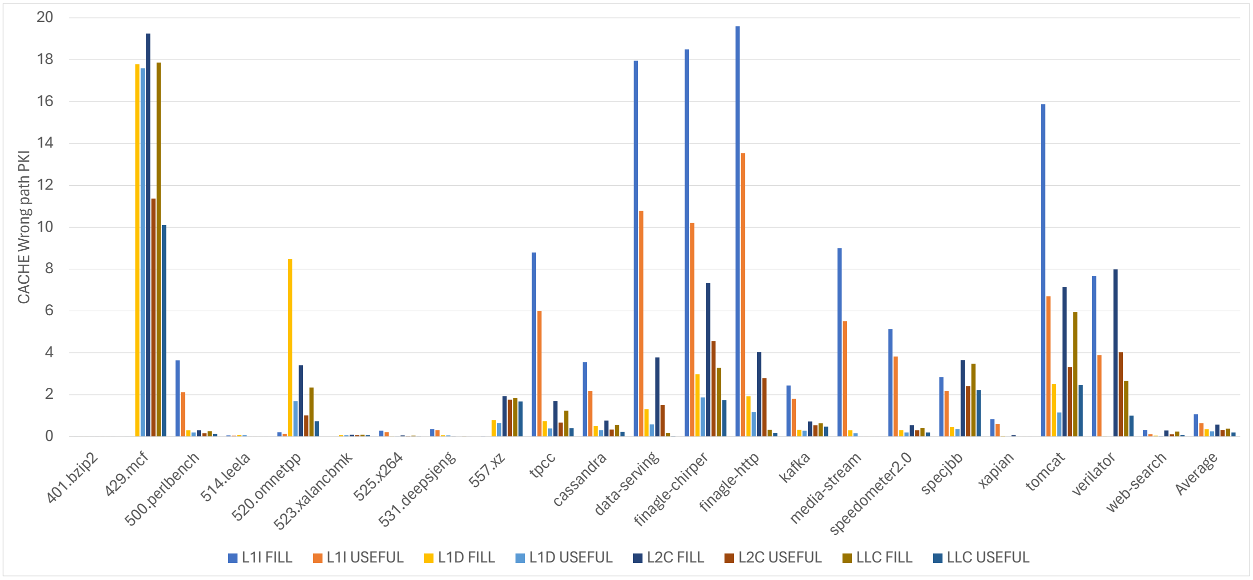


Figure 2.8: Useful Wrong Path lines in Per Kilo Instruction (PKI) at L1, L2 and L3 caches

Cache lines brought in during the wrong path execution may either prove useful or useless. Wrong path lines that are eventually utilized in the correct path later contribute positively to performance improvement, whereas those evicted without being used may lead to performance degradation, particularly if useful cache lines are displaced.

Figure 2.7 illustrates the percent performance gain of the wrong path model over the correct path model for various SPEC17 and datacenter workloads. Most benchmarks exhibit performance improvements attributed to the subsequent utility of wrong path lines. Performance gains range from -1.7% (e.g., xapian) to as high as 22.4% (e.g., 557.xz). This performance variation falls within the range of EMISSARY-L2 and PDIP, highlighting the necessity of employing execution-driven models for evaluating front-end enhancement techniques.

Cache lines brought in during the wrong path may prove useful later in the program's correct path execution, thereby potentially enhancing overall performance. Figure 2.8 illustrates the total useful lines and fills at various cache levels attributable to wrong path execution, expressed as per kilo instructions (PKI). The ratio of useful lines to fill lines indicates the usefulness of wrong path lines. Benchmarks with higher useful PKI tend to exhibit greater performance improvement. On average, approximately 60% and 68% of wrong path fills in the instruction cache and data cache, respectively, are deemed useful. Consequently, the overall net impact of wrong path instructions tends to be positive.

Figure 2.9 illustrates the percentage increase in I-Cache MPKI when the Wrong Path is enabled, compared to the Correct Path model with FDIP, across various SPEC and datacenter workloads. Notably, benchmarks like `tomcat` and `web-search` exhibit MPKI increases of over 100%. On average, there is a 32.8% increase in I-Cache MPKI. This highlights the imperative for front-end optimizations to incorporate FDIP with wrong path modeling to ensure accurate simulation results. Consequently, we implemented FDIP within the execution-driven Out-of-Order CPU model of the gem5 simulator, which models wrong path effects.

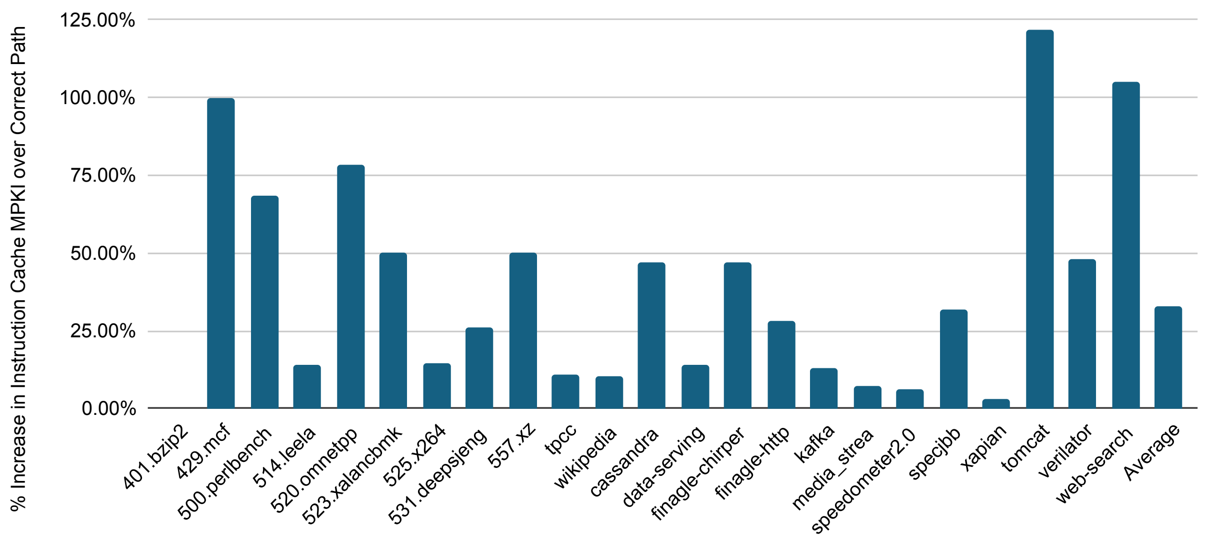


Figure 2.9: % change in Instruction Cache MPKI with FDIP and Wrong Path model over Correct Path Model

2.4 Performance of FDIP

FDIP model described in Section 2.1 is faithfully modeled execution-driven Out-of-Order (OOO) CPU model of the gem5 simulator. Figures 2.10, 2.11 and 2.12 show percent improvement in the IPC of ARM, x86, and SEPC x86 workloads over baseline without FDIP. The ARM workloads show higher improvement in performance compared to x86 workloads due to high front-end pressure. Additionally, some optimizations like Early Prefetch correction were enabled only for the ARM ISA model. Overall geomean IPC gains are 35.8%, 19.6%, and 14.7%, respectively. This shows that our FDIP model is very aggressive, and our performance gains corroborate with Ishii et al. [41]. The FDIP model is used as the baseline in the dissertation unless stated otherwise.

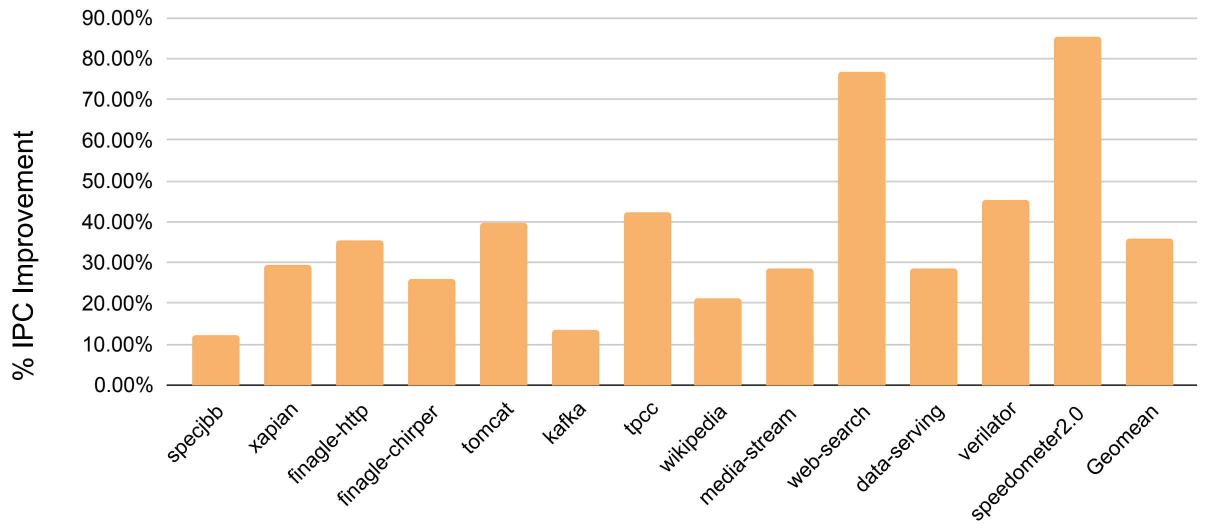


Figure 2.10: % IPC gain of ARM datecenter workloads with FDIP over No FDIP model

2.5 Not all misses are equal

The design goal of FDIP is to tolerate I-Cache misses. Similar to the Out-of-Order execution to hide latencies of data cache misses in the backend, FDIP hides latencies of I-Cache misses. To demonstrate misses have unequal costs, consider the example shown in Figure 2.13. In this example, a four-entry FTQ is used. The latency of the I-Cache hit is two cycles, and the L2 hit is a minimum of eight cycles. The state of FTQ is shown at the end of each cycle. There are two operations happening every cycle. FTQ is fed a new entry every cycle by the BPU, and the prefetch engine issues one prefetch request for every cycle. Assume

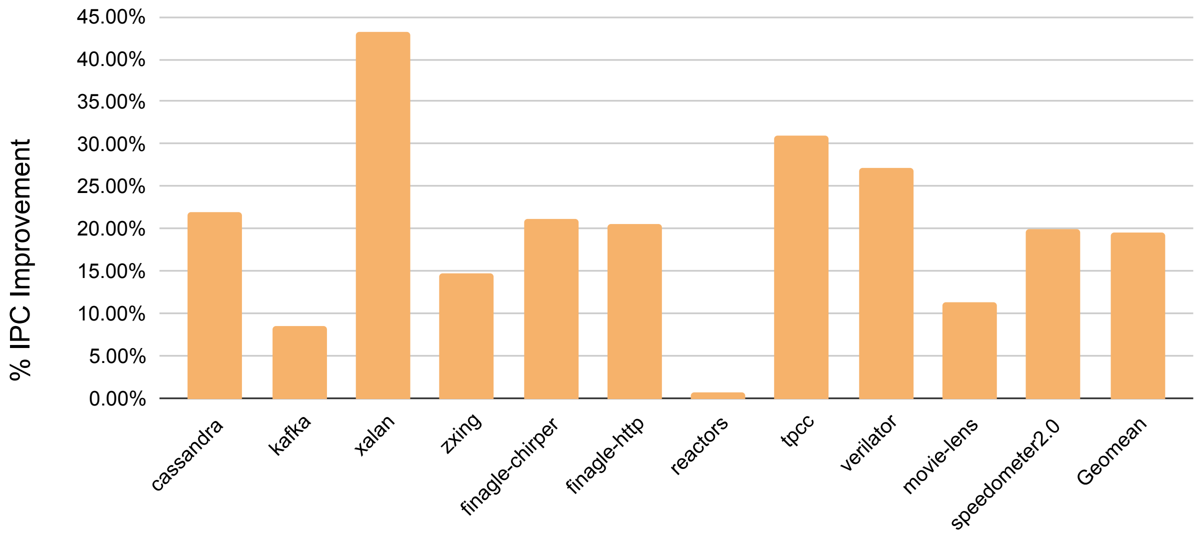


Figure 2.11: % IPC gain of x86 datecenter workloads with FDIP over No FDIP model

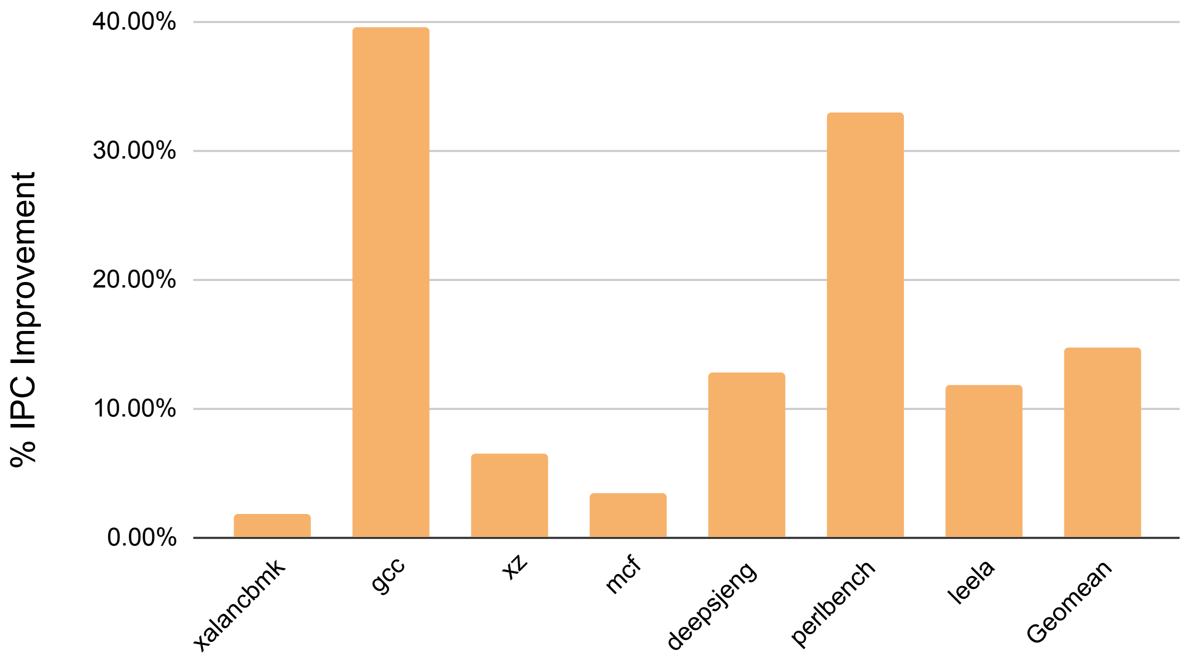


Figure 2.12: % IPC gain of x86 SPEC'17 workloads with FDIP over No FDIP model

that FTQ has one entry $R0$ at the beginning of cycle 0 and no prefetch requests have been issued yet. In cycle 0, the prefetch engine issues a prefetch request to I-Cache for R0. In the same cycle, BPU feeds FTQ with entry $R1$. Similarly, in cycle 1, a prefetch request for cache line $R1$ is issued, and a new entry $R2$ is inserted. At the beginning of cycle 2, the $R0$ is found to miss in the I-Cache so the request is forwarded to L2, and a prefetch request for $R2$ is issued. FTQ is also populated by $R3$ in cycle 2. Similar operations are repeated till the end. In cycle 5, request $R3$ is found to miss in I-Cache, so the request is forwarded to L2. $R0$ and $R3$ both have missed in I-Cache, but the effective miss cost is ten cycles and one cycle, respectively. The FTQ effectively hid the latency of $R3$ under the shadow of $R0$ miss, thereby tolerates I-Cache miss latency. Thus, not all misses are equal. A sufficiently large FTQ can hide I-Cache miss latency completely without any prior miss in the FTQ.

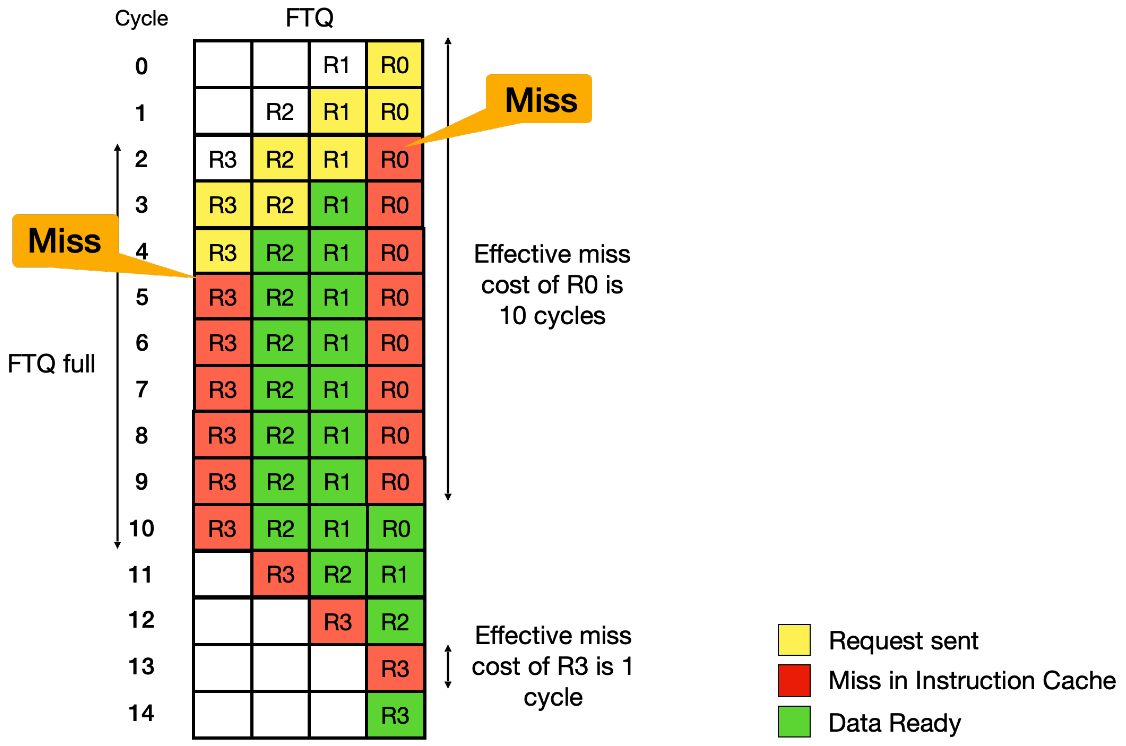

Cycle
FTQ
Miss
Miss
FTQ full
Effective miss cost of R0 is 10 cycles
Effective miss cost of R3 is 1 cycle
Request sent
Miss in Instruction Cache
Data Ready

Figure 2.13: Pipeline diagram of FTQ processing requests R0 to R4. State of FTQ shown at the end of each cycle from cycle 0 to cycle 14.

2.6 Impact on Criticality

An important component in a decoupled front-end machine is the depth of the FTQ, which determines how far the predicted instruction stream (which drives the prefetching) can get ahead of the actual fetch demand. A sufficiently large FTQ offers a deep enough instruction prefetching window such that a full L1I miss, and possibly even an L2 miss, can be tolerated without stalling the IFU. This assumes the FTQ is full, given that the BPU nearly always sustains a higher throughput than the backend, the FTQ is generally full in the absence of FTQ resetting events (e.g., branch mispredicts). Modern decoupled front-end processors, therefore, implement deep FTQs to get the most benefit from prefetching and tolerate cache miss latency. The importance of FTQ in the context of instruction prefetching was amply discussed in [40]. The authors show that most of the performance benefits obtained with recent instruction prefetching proposals [69] vanish with a decoupled front-end machine implementing a modest 24-entry FTQ. The key insight here is that in the presence of FDIP, which has the potential to hide the full latency of the majority of misses, we see high variance in the performance-criticality of L1I misses. Some misses have no impact on front-end (or, naturally, backend) performance, while others still do. Thus, increasing I-Cache size does not always lead to a significant increase in performance, even though I-Cache misses are the major source of stalls. This dissertation aims to improve front-end performance by focusing on critical instruction cache misses instead of all cache misses. A line is considered critical when it meets the following conditions: (1) the line must have retired an instruction, (2) the line must have missed the instruction cache, and (3) the line must have produced front-end stalls as a result of the miss.

Chapter 3

Criticality-Aware Cache Replacement Policy for Datacenter Workloads

Observing that FDIP-enabled front-end can tolerate instruction cache (L1I) misses prior work introduced first criticality-aware instruction policy EMISSARY (Enhanced MISS-Awareness Replacement Policy) [57] to preserve instruction lines in L1I. EMISSARY prioritizes lines that caused decode to starve in the L1I cache. This policy works well for workloads that have high L1I MPKI but low L2 Instruction MPKI. Since EMISSARY works at L1I, it is referred to as EMISSARY-L1. Datacenter workloads have code footprints much larger than the size of the L3 cache. EMISSARY-L1 policy is ineffective when applied to the L1 cache due to the high volume of criticality lines in datacenter workloads. An aggressive front-end fails to completely hide latencies of L2 cache misses. Datacenter workloads have very high L2 cache instruction misses. The L2 cache is shared between data and instruction lines, due to which the EMISSARY-L1 [57] is not directly applicable to the L2 cache. Prioritizing instruction lines in a unified cache like L2 requires carefully balancing instruction and data lines. In order to address these challenges, a new, improved EMISSARY-L2 is proposed in this chapter. EMISSARY-L2 makes two key improvements over EMISSARY-L1. First, it proposes an improved criticality filter, which helps in identifying cache lines that also impact the backend of the processor. Two, improved management of critical cache lines in the L2 cache. This chapter first introduces the original EMISSARY-L1 and then proposes new improvements.

3.1 Original EMISSARY Policy

This section describes the prior EMISSARY-L1 [57] policy, which was proposed for cache priority lines in the L1I cache[1]. The key idea is that only lines that caused starvations will likely cause more starvations in the future. An EMISSARY-L1 cache leverages this by holding on to starvation-causing lines for longer, even if they have been less recently accessed than other starvation-free lines. Thus, EMISSARY-L1 cache replacement policies are bimodal. Bimodal techniques have two orthogonal aspects: *mode selection* and *mode treatment*. These aspects are discussed in this section.

3.1.1 Mode Selection

The two modes of a bimodal cache replacement policy are referred to as *high* and *low* priority, respectively. Table 3.1 shows the mode selection options for the space of realizable cache replacement algorithms referenced in this chapter. These mode selection options are combined in Boolean equations. For example, S&R(1/32) requires a missed line to have caused starvation (S) during the miss AND to have been the lucky one of 32 chosen by pseudo-random selection (R(1/32)). EMISSARY policies all contain S in their mode selection equations. For all policies in this chapter, the mode selection is determined once during cache line insertion. LRU can be thought of as a bimodal predictor degenerated to treat all inserted lines as high-priority by MRU position placement.

[1]Much of this content is common with [58], a work with co-authors

Notation	**Description**
1	Always High-Priority
0	Never High-Priority
R(r)	High-Priority with random probability r
S	High-Priority, line miss causes decode starvation

Table 3.1: Mode Selection Options

Notation	**Description**
M	Insert High-Priority lines in MRU position, otherwise LRU
P(N)	Protect up to N MRU High-Priority lines/set from eviction

Table 3.2: Mode Treatment Options

3.1.2 Mode Treatment

A meaningful bimodal cache replacement policy must treat lines differently based on the selected mode. Thus, the next aspect determines how high-priority lines are treated differently from low-priority lines. All realizable cache replacement policies discussed here use one of the two bimodal behaviors shown in Table 3.2.

In the first, M, bimodality comes from inserting high-priority lines into the cache's MRU position while placing low-priority lines into the cache's LRU position [65]. In the second, P(N) is the EMISSARY-L2 behavior. It is described by Algorithm 2. P(N) techniques do not act on priority at insertion. Instead, the priority is recorded as a priority bit (P) associated with each line that impacts eviction. High-priority lines have $P = 1$, while low-priority lines have $P = 0$. When inserting a line L into a P(N) cache, if the number of high-priority lines in the set is less than or equal to the maximum N if it is then the line L to be inserted (regardless of priority) replaces the LRU of the low-priority lines. Thus, the step in line 2 may increase the number of high-priority lines in the cache but cannot reduce it. For insertions where the number of high-priority lines in the set is greater than the maximum N, the cache evicts the LRU among the high-priority lines. Note that the number of high-priority lines are not reduced to less than N at any point without a separate reset mechanism.

Algorithm 2 The EMISSARY-L2 Eviction Policy

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

The EMISSARY-L1 treatment option is orthogonal to the specific LRU algorithm used. For lines 2 and 4 of Algorithm 2, finding the LRU among the low-priority or the high-priority lines can be calculated precisely from a true LRU algorithm. With a pseudo-LRU (PLRU) algorithm, however, keeping separate PLRU's for low- and high-priority lines limits the imprecision. The PLRU-based EMISSARY-L1 uses the Tree Pseudo-LRU (TPLRU) algorithms with separate trees for low- and high-priority lines. When a high-priority line is accessed, only the high-priority tree is updated. Likewise, for a low-priority line and tree. For eviction, the appropriate tree is used to find the line to replace, skipping any lines that do not match the priority criteria. TPLRU requires $ways - 1$ bits per tree. The evaluations use the TPLRU implementation.

3.2 Improved EMISSARY

The EMISSARY-L1 considers a cache line to be critical only when the decode stage is starved (S). This criterion is further combined with random probability (R) to reduce single or low-reuse critical lines. However, EMISSARY-L1 is susceptible to prioritizing cache lines in the wrong path as it updates priority entries speculatively. Workloads with high branch misprediction rates could eventually prioritize cache lines that are in the wrong path, thereby reducing effectiveness.

To address this issue, the improved EMISSARY considers cache lines that are retired. A front-end critical line may not always improve performance, especially if the backend has sufficient work or is waiting for long-latency data cache misses. Improved EMISSARY-L2 takes into account cache lines that would genuinely contribute to improving overall performance. This is achieved by considering lines only when the Issue Queue is empty (E).

Overall, a line is considered critical when the following conditions are met:

- An instruction cache line has retired at least one instruction.
- A cache line that missed in the Instruction Cache.
- It caused decode starvation cycles when the backend is idle.

The EMISSARY-L2 follows a new eviction policy as described in Algorithm 3. When the number of priority lines has not reached its limit (N), the eviction policy is the same as EMISSARY-L1. However, when the number of high-priority lines reaches the limit N, EMISSARY-L2 demotes one of the high-priority lines instead of retaining priority lines longer to make room for the new line being promoted.

Since L2 is shared between data cache lines and instruction cache lines, preserving high-priority lines when there is contention for data cache lines is detrimental. Additionally, as the access rates of L2 are much lower than that of L1, priority lines that stay for a very long time need to be evicted. Evicting stale priority lines to make room for new priority lines alleviates the problem of priority lines becoming stale to some extent but not completely. To further address this concern, the priority bits of all cache lines must be reset after a certain duration. Empirically, we found that resetting the priority of all cache lines after 100M instructions strikes a good balance in preserving priority lines long enough to be reused before they become stale.

Algorithm 3 The EMISSARY-L2 Eviction Policy

```
1: if number of high-priority ($P = 1$) lines <= $N$ then
2:     Evict the LRU among the low-priority ($P = 0$) lines
3: else
4:     Evict the LRU among high-priority lines
5: end if
```

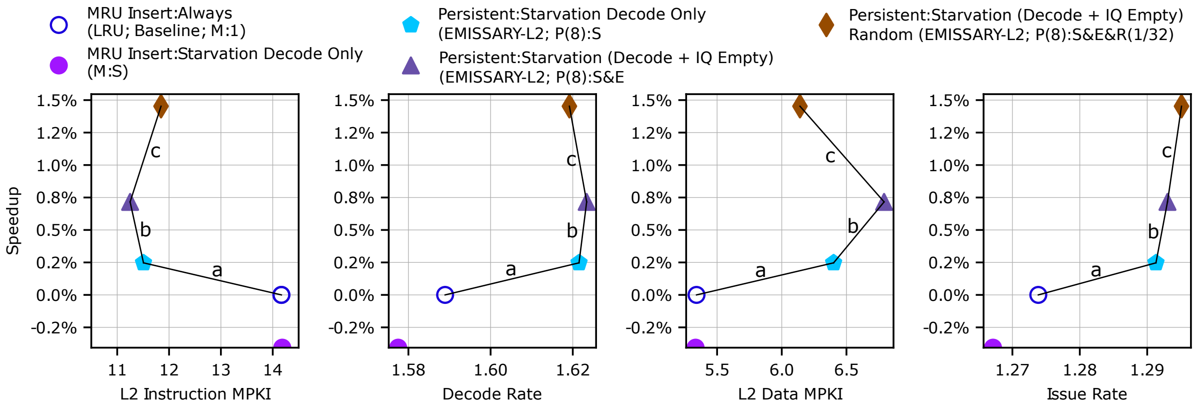


Figure 3.1: Speedup vs L2 Instruction MPKI, Decode Rate, L2 Data MPKI, and Issue Rate of various cache replacement policies for `tomcat` benchmark on a 1M 16-way L2 cache(with true LRU and no prefetchers).

3.2.1 Impact of Issue Queue Empty signal

The EMISSARY-L2 policy is applied to the L2 cache instead of the L1 cache proposed by EMISSARY-L1. Datacenter workloads typically exhibit much larger code footprints than the size of the L2 cache, resulting in very high L2 MPKI (misses per kilo-instructions). On average, the L2 MPKI exceeds 10. Given that the L2 cache has orders of magnitude higher capacity than that of an L1 cache, EMISSARY-L1 would be ineffective on datacenter workloads because the L1's capacity is not sufficient to hold onto priority lines long enough before they are reused. (A more detailed analysis of datacenter workloads, including reuse distance, is presented in section 3.3). Therefore, EMISSARY-L2 is applied to the L2 cache.

The L2 cache serves both data and instruction lines, unlike L1I, which is exclusively for instruction cache lines. Preserving cache lines in the L2 for instruction lines could be detrimental when workloads have higher data footprints. Therefore, it is crucial to consider cache lines that not only experienced decode starvations but also have an impact on the backend of the processor. Figure 3.1 illustrates the performance of various bimodal policies, along with their impact on L2 MPKI (Instruction and Data), decode rate, and issue rate.

In Figure 3.1, policies with increasing levels of criticality filtering are connected by trend lines labeled *a*, *b*, and *c*. The EMISSARY-L2 policy effectively allocates a portion of the L2 cache for instruction cache lines, which impacts data misses. It is evident across all policies that L2 Data MPKI is higher than the baseline.

The EMISSARY-L2 with decode starvation-only policy, serving as a proxy for EMISSARY-L1, demonstrates improvement over the baseline when applied to L2. The additional benefit of the Issue Queue Empty signal is evidenced by higher performance. However, the L2 data cache MPKI is higher with the Decoder and Issue Queue Empty policy, primarily due to low-frequency lines being prioritized.

When the Issue Queue Empty signal is further combined with randomness, improved performance is observed, attributed to preventing low-reuse lines from being promoted. This can be observed by the reduction in L2 data MPKI. Thus, the Issue Queue Empty signal emerges as a key factor in demonstrating the improved performance of EMISSARY-L2.

3.3 Decode Starvation Behavior of Datacenter Workloads

Discussion in this section is borrowed from shared co-authored work EMISSARY.Instruction fetch is responsible for keeping the decode stage fed. If the processor could perfectly predict the target of every control-flow instruction, instruction fetch could issue all of its memory requests early enough to tolerate the latency to main memory without starving decode. Unfortunately, even the best branch predictors are not perfect. They are, however, quite good. Modern processor front-ends incorporate decoupled, aggressive fetch engines guided by excellent branch predictors, large BTBs, and pre-decoders [40]. Such front-ends accurately fetch several tens or even hundreds of instructions early. Instruction decode queues filled this way can often tolerate L1I misses before emptying and leading to decode starvation; this is especially true when an L1I miss leads to an L2 hit. From the perspective of cost-aware cache replacement policies, keeping lines with tolerated L1I misses in the cache has less utility than keeping lines whose misses cause decode starvation. The key to making this work involves differentiating tolerated L1I misses from those that cause starvation.

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

Beyond branch prediction, many factors interact to determine whether or not an L1I miss will cause decode starvation. For example, a *stalled* decode cannot *starve*. Decode stalls occur when the processor back-end cannot accept more instructions. When decode stalls, it does not attempt to pull from the instruction queue, a necessary condition for starvation.

While predicting starvation by its component factors is hard, observing starvation during execution is easy. Existing signals assert when starvation occurs (likewise when the issue queue is empty). Furthermore, when decode starvation occurs, the address for which the decode is waiting is also known. All of this information is known many cycles before the line that missed under these circumstances is inserted into the cache. Knowing this information in advance does not mean that it is necessarily of value to a cost-aware cache replacement policy.

An instruction fetch that causes starvation must be a miss in the L1I cache. These L1I misses could be served either from L2 cache, L3 cache, or main memory. Lines served from the L2 cache introduce fewer starvation cycles than the ones served from more remote levels.

The first bar in Figure 3.2 depicts the distribution of reuse distance based on observed cache line accesses in the committed path across the datacenter workloads used in this study. Reuse distance is measured as the number of unique lines accessed between two access to the same line. The same line accessed consecutively is not counted. Reuse distances are categorized into three buckets - Short [0-100), Mid [100-5000), and Long [>5000) Reuse. Short Reuse lines are likely to hit in L1I, Mid Reuse lines are likely to miss in L1I and hit in L2, and Long Reuse lines are likely to miss in L2. This predicted behavior is confirmed in the second bar showing the % of Long Reuse accesses that miss in L2. Overall, more than 90% of L2 misses are from Long Reuse lines.

The third bar in Figure 3.2 shows the interplay between these different categories of reuse lines and decode starvation. Interestingly, more than 90% of the starvation cycles are caused by Long Reuse lines, which account for less than 20% of all accesses. Thus, a small number of accesses contribute to the majority of starvation cycles – a property that can be utilized by a replacement policy. Consequently, EMISSARY-L1 at the L1I cache may have little value as the lines that cause the majority of starvations are Long Reuse lines, lines which the L1I cannot realistically preserve. Since the majority of starvations are caused by L2 miss lines, in this work, EMISSARY policies are applied only to instructions cached in the L2 cache. The L1I does play a role in that only L1I misses causing starvation are treated as high-priority. **A line's priority is only communicated to L2 cache once it is evicted from the L1I cache.** Instruction lines cached in L2 are then guided by the EMISSARY-L2 replacement policy. In this way, Long Reuse lines that have caused starvations are now cached in the L2 cache for longer. EMISSARY-L2 uses 2 bits per line to record priority and for TPLRU access in the L1I and L2 caches. It is 1024 bits in L1I (32kB cache with 64B line size) and 32,768 bits in L2 (1MB cache with 64B line size), a little over 4kB total.

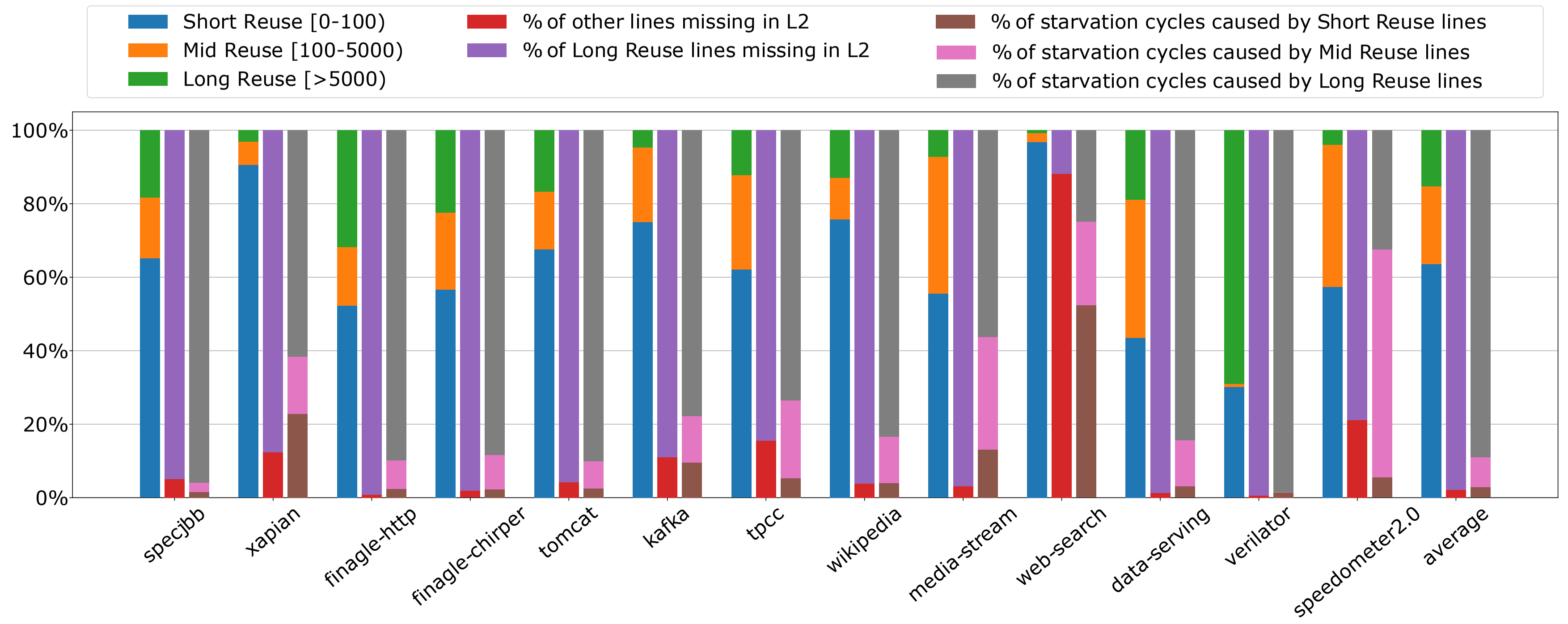


Figure 3.2: First bar: distribution of Short, Mid, and Long Reuse access lines. Second Bar: fraction of L2 Instruction Misses by Long Reuse lines. Third Bar: distribution of starvation cycles caused by Short, Mid, and Long Reuse lines.

Notation	Description
M:1	Always insert as MRU; Classic LRU; Baseline
M:0	Never insert as MRU (only as LRU); LRU Insertion Policy (LIP) [65]
M:R(r)	MRU insert with probability r; Bimodal Insertion Policy (BIP) [65]
M:S&E	MRU insert when starvation occurs and issue queue is empty
M:S&E&R(r)	MRU insert when starvation occurs, issue queue is empty and with probability r
P(N):R(r)	EMISSARY-L1 bimodal behavior only; high-priority lines selected with probability r
P(N):S	EMISSARY-L1 [57]: high-priority on starvation
P(N):S&E	Improved EMISSARY-L2 : high-priority on starvation and empty issue queue
P(N):S&E&R(r)	Improved EMISSARY-L2 : high-priority on starvation, empty issue queue, and with probability r
SRRIP	Static re-referrence interval prediction [43]
BRRIP	Bimodal re-referrence interval prediction with probability (1/32)[43]
DRRIP	Dynamic re-referrence interval prediction [43]
PDP	Static protective distance policy [30]
DCLIP	Dynamic Code Line Preservation [42]

Table 3.3: Cache replacement policies explored

3.4 Cache Replacement Policies

The top of Table 3.3 shows the prior work and proposed bimodal cache replacement policies used in this work. Each policy is a combination of a mode selection option (individually or by combination with a Boolean expression) and a mode treatment option described earlier. The rest of Table 3.3 lists other advanced policies used in the experimental comparison.

3.5 Experimental Exploration

This section describes the simulation infrastructure, machine model, and benchmarks used to evaluate EMISSARY-L2 in various ways[2].

[2]Much of this content is common with [58], a work with co-authors

3.5.1 Simulation Infrastructure and Machine Model

This study uses gem5, a popular cycle-accurate simulator [27], running a detailed CPU model in FS (Full System) mode with a full Operating System (OS). Gem5 supports a checkpointing mechanism that creates reusable snapshots for later restarts. For datacenter workloads, collecting gem5 checkpoints itself can be a significant bottleneck in evaluating microarchitectural changes. To reduce this, we used the QEMU [26] emulator and built a tool, QPoints (see 5.1.3), to create gem5-compatible snapshots. Snapshots consist of a dump of the physical memory, disk image, device state, CPU architectural state, and a checkpoint file compatible with gem5. Typically, checkpoint files can only be ported to another environment with the same system board configuration. We extended gem5 to support a hardware board configuration called `virt_machine` for snapshots created with QEMU. This enables the use of QEMU fast emulation features, like hardware acceleration with KVM [39]. We used an ARM64 system running Ubuntu 18.04 with Linux kernel 4.15 and an Apple M1 Mac mini to accelerate QEMU emulation.

As shown in Table 3.4, we used Intel's Alderlake-like model for all experiments with the TPLRU config. The next line prefetcher (NLP) is enabled in the Adlerlake-like model for the L1D, L2, and L3 caches. The baseline for all experiments has FDIP enabled. L3 is an exclusive cache with DRRIP. L2 uses an SFL (Served From Last-level) bit to track each line's origin (i.e., L3 or memory). When a line with its SFL bit set is evicted from L2, it is placed at the MRU position in the L3. Each simulation includes a warm-up period of 5 million instructions from the resumed state, followed by a measurement period of 100 million instructions in the detailed simulation model.

3.5.2 Decoupled Fetch Engine

We extended the fetch engine of the gem5 simulator to model the aggressive front-end found in modern processors with state-of-the-art FDIP prefetchers [67]. FDIP includes a Fetch Target Queue (FTQ) to decouple the fetch pipeline from the rest of the processor, enabling the fetch pipeline to run ahead of the rest of the processor pipeline [40, 68, 67]. The fetch pipeline, including the BTB and FTQ, has been modified to operate at the dynamic basic block granularity.

We modified the BTB so that each entry corresponds to a basic block. In addition to the target, entries contain details pertaining to the basic block - starting address, size, and the type of control-flow instruction that ends the basic block. This enabled the BTB to be indexed based on the branch target or the basic block's starting address instead of the branch PC. We used ARM binaries in this work. ARM's fixed-length encoding made it easier to model the BTB. Specifically, given the starting address and size of the basic block in terms of the number of instructions, it is straightforward to find the address of the control-transfer instruction that ends the respective block. This flexibility helped in reducing the otherwise necessary changes to the branch predictor. Variable-width instructions can be supported with additional hardware.

The branch predictor and BTB enqueue up to one basic block prediction per cycle to the FTQ. As in the BTB, each entry in the FTQ contains the starting address and size of the dynamic basic block. Naturally, enqueuing stalls on BTB misses. The next two fall-through lines are prefetched in the event of a BTB miss. As an enhancement, the modeled front-end also includes a pre-decoder to update the BTB and minimize such enqueue stalls proactively. Branch re-steers flush the FTQ before resuming predictions on the corrected path. The FTQ along with basic-block level fetch enabled the front-end to run-ahead from the processor pipeline soon after every flush operation.

The FTQ enhancements allowed memory requests to be issued early. This work includes an FTQ size of 24 entries with a 192-instruction buffer. This offered the right balance by having sufficient starvation tolerance for hiding many L1I misses (see §3.3) while keeping the front-end from becoming overly aggressive in the presence of branch mispredictions. The extended run-ahead front-end requires the instruction buffer to be able to receive memory

Field \ Model	Alderlake-like
ISA	Aarch64 (64-bit ARM)
Private L1I, L2D Caches	32kB (I), 64kB (D) NLP, 8-way 64B line size, 2 cycle hit TPLRU
Unified L2 Cache	1MB, 16-way, 64B line size 12 cycle hit, Inclusive NLP
Shared L3 Cache	2MB, 16-way, 64B line size 32 cycle hit latency Exclusive Victim Cache NLP DRRIP + SFL
Branch Predictor	TAGE, ITTAGE
BTB size	16K entries
Fetch Target Queue	24 entry 192-instruction
Fetch/Decode/ Issue/Commit	8 wide
ROB Entries	512
Issue/Load/Store Queue	240 / 128/ 72
Int/FP Registers	280 / 224

Table 3.4: Processor configurations

responses out-of-order. Overall, our optimized FDIP provides a geomean speedup of 33.1% over a no FDIP model for the 13 datacenter benchmarks as described in Section 3.5.3.

3.5.3 Benchmarks

To evaluate EMISSARY-L2 , we used 13 popular server applications with large code footprints from various benchmark suites: `tomcat` (Apache's implementation of Jakarta Servlet, Jakarta Expression Language, and WebSocket [4], from Dacapo benchmark suite [28]); `kafka` (Apache's distributed event streaming application used by companies like LinkedIn [2], from Dacapo benchmark suite); `tpcc` (On-Line Transaction Processing workload [14], from OLTP-Bench suite [29]); `wikipedia` (MediaWiki application on Wikipedia dataset [8], from OLTP-Bench suite); `data-serving` (Cassandra NoSQL database application [1], from Cloudsuite V4 [32]); `media-streaming` (Simulates video traffic, from Cloudsuite V4); `web-search` (Apache Solr search engine application [3], from Cloudsuite V4); `xapian` (a web-search application, Tailbench suite [46]); `specjbb` (A SPEC benchmark to test Java application features [12], from Tailbench); `finagle-http` (Twitter's HTTP server [15], from Renais-

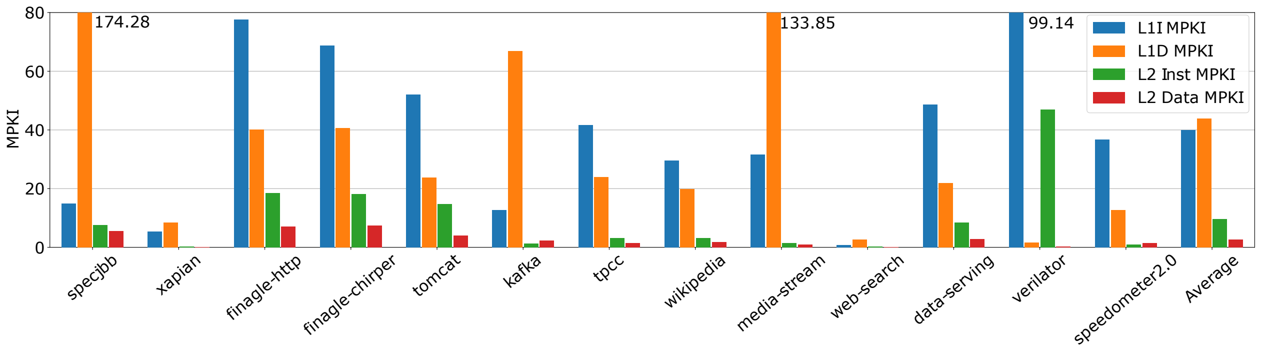


Figure 3.3: Average L1I, L1D, L2 Instruction, and L2 Data Cache MPKI of the benchmarks on the TPLRU + FDIP baseline.

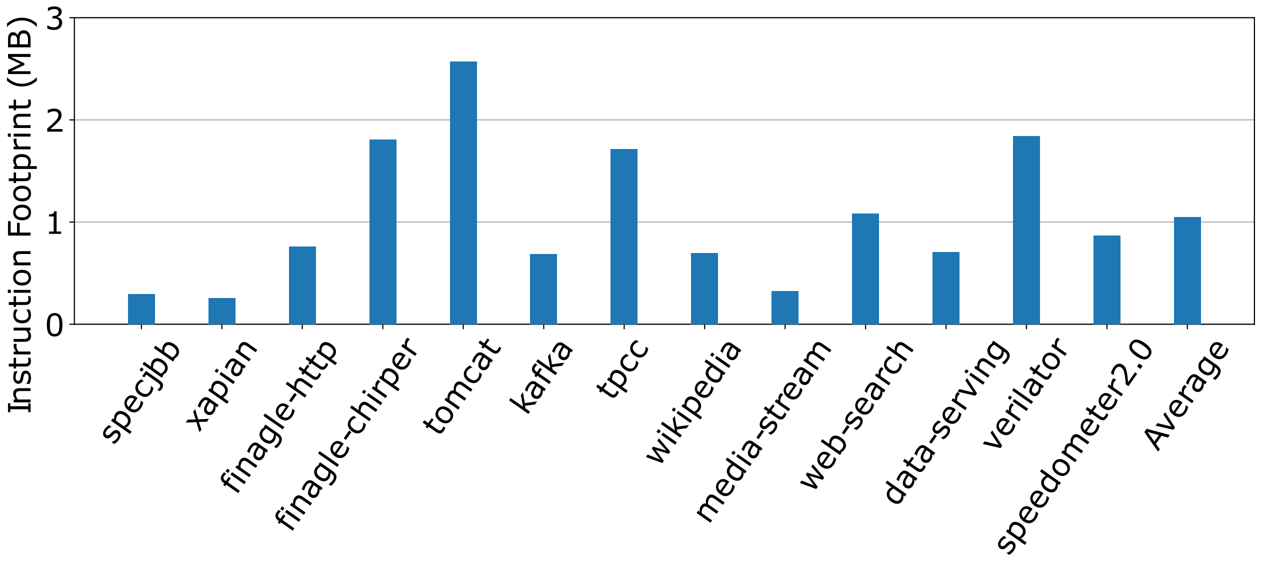


Figure 3.4: Instruction footprint of all benchmarks

sance [64]); `finagle-chirper` (A microblogging service by Twitter, from Renaissance); `verilator` [16] (simulates the RTL design of Rocket Chip [24] simulating quick sort code); and `speedometer2.0` (a Java Script benchmark runs on a web browser benchmark which tests for the number of threads spawned in a minute [13]).

Benchmarks from the Tailbench suite are compiled using the default flags provided by the suite. `verilator` benchmark was built from the code provided and also optimized further using Facebook's BOLT [61] binary optimization tool. All benchmarks were built on the emulated environment described in Section 3.5.1. `speedometer2.0` benchmark is simulated on a Chromium web browser.

Since all benchmarks except `verilator` are multithreaded, we scaled them to a single core ($N = 1$) for the evaluation on the simulator. To ensure that this simulation of a multithreaded workload is meaningful, we looked for performance trend differences between single core ($N = 1$) and multicore ($N > 1$) thread scalings on a real x86 Linux host machine using hardware performance monitoring counters. We examined the data at the feature level (e.g., branch misprediction rate), at the overall performance level, and according to the methodology outlined in [79]. We determined with confidence that the single-core scaling of these applications had the same workload characteristics as the $N = 4$ and $N = 8$ scalings. Software thread scheduling during simulation is handled by the Linux thread scheduler in Full System mode.

The benchmarks used exhibit various characteristics, as shown in Figure 3.3. `specjbb`, `kafka`, and `media-stream` have very high L1D MPKI when compared to L1I MPKI. The average L1D MPKI is higher than the average L1I MPKI. `media-stream` and `kafka` benchmarks additionally have a higher L2 Data MPKI than L2 Instruction MPKI. However, the average L2 Instruction MPKI (9.63) is much larger than the Data counterpart (2.69). Figure 3.4 shows the instruction footprints of all benchmarks. Instruction footprints are measured based on the total number of unique cache lines accessed by the application during the simulation times the cache line size. `tomcat` has the highest footprint of 2.57 MB and `xapian` has the lowest footprint of 0.29 MB. The average footprint of the selected workloads is 1.05 MB. The chosen workloads were selected over the more traditional SPEC CPU workloads because they have larger code footprints and do not easily fit into the larger L2 caches of modern processors. Also, these benchmarks have been used in many related works as well [51, 49].

3.5.4 Policy Selection and Parameterization

Section 3.1 outlines a large space of possible cache replacement policies. To narrow the design space to a small and meaningful set of policies, using an initial exploration, we first

P(N)	S&E	R(1/2)	R(1/8)	R(1/16)	R(1/32)	R(1/64)	S&E&R(1/2)	S&E&R(1/8)	S&E&R(1/16)	S&E&R(1/32)	S&E&R(1/64)	# Best
2	-0.350	-1.511	-0.460	**0.216**	0.053	0.166	0.116	0.947	0.969	1.245	**1.548**	0
4	1.946	-0.433	0.736	**1.337**	1.171	0.897	1.621	1.767	2.025	**2.379**	1.634	0
6	1.995	0.7	1.406	1.571	**2.023**	1.813	1.656	2.261	2.486	**2.546**	1.906	5
8	1.294	0.579	0.906	1.112	**1.354**	1.193	**2.576**	2.301	2.419	2.490	2.005	2
10	-0.020	-0.995	-0.247	-0.287	-0.173	**-0.066**	0.125	1.47	**2.507**	3.15	2.018	1
12	-3.275	-4.926	-4.072	-3.751	-2.927	**-1.834**	-2.378	2.07	2.153	**3.063**	2.235	0
14	-7.698	-10.941	-8.710	-7.269	-5.472	**-3.628**	-5.039	-0.087	1.878	**3.241**	2.385	2
# Best	-	0	0	2	2	3	1	0	1	4	1	-

Table 3.5: Geomean speedup with respect to a LRU + FDIP baseline (Alderlake model) across all configurations for various values of r and N when run on a system with EMISSARY-L2 Policy at L2 Cache

select a small set of desirable policy types and then find a reasonable set of configuration parameters for these policy types. The useful representative policy types chosen are the ones listed in Table 3.3. Ideally, we would like to find a single value of r and N that works well across all policies. Based on prior work, we expect the best r to be from 1/2 to 1/64 [65]. For a 16-way cache, useful values of N are from 2 to 14.

Table 3.5 shows the geometric mean speedup across all programs for a ranch of r and N values. The "#Best" row (or column) indicates the number of best configurations found in each column (or row). An r value of 1/32 consistently gives the best results in many cases. Prior work (M:R(r), BIP [65]) also suggests 1/32 or 1/64 as the value for r. Generally, benchmarks reach peak performance when N is 8, except `verilator`, which continues to improve as N goes to 14. Hence, we set $N = 8$ and r =1/32 for the evaluation.

3.5.5 Performance

Figure 3.5 shows the speedup versus MPKI (odd rows) and speedup versus change in starvation cycles (Decode + IQ Empty) for committed instructions (even rows). For space reasons, `tpcc` is omitted as its L2 instruction MPKI is quite low. Values of N shown range from 0 to 14 by 2. An N of 0 is equivalent to the baseline. Lines connect P(N) to P($N + 2$) for each N from 0 to 12.

Generally, when MPKI is greater than 1.0, performance increases, and starvations reduce as N increases to 8 (i.e., half of the 16-way cache is preserved for high-priority instruction lines). As N increases further, the performance gains decrease despite the consistent starvation reduction. This is because the L2 cache is shared by instructions and data. As more ways get used by high-priority instruction lines, resources are constrained to data lines, leading to more back-end stalls. See §3.5.8.

The results show that higher performance can come without much change in MPKI. This is the central observation of all cost-aware cache replacement policy proposals, and this observation is confirmed for the EMISSARY-L2 techniques. Not all cache misses in modern out-of-order processors have the same cost. A significant portion of the misses can be tolerated without degrading processor performance. Similarly, a significant portion of the addresses that are latency-sensitive and cause decode starvations do so every time they are accessed. EMISSARY-L2 handles both of these categories very efficiently. It assigns higher priority to starvation-prone addresses, keeping them in the cache longer even if they are not accessed frequently. EMISSARY-L2 gives lower priority to latency-tolerant addresses, but it does cache them long enough to capture as much of their (belated) temporal locality.

The speedup and energy reduction of EMISSARY-L2 compared to other techniques over the TPLRU baseline is shown in Figure 3.7. Overall, P(8):S&E&R(1/32), the preferred EMISSARY-L2 configuration, yields a geomean speedup of 2.49% on all benchmarks, with gains as high as 11.7% (`verilator`) and as low as -1% (`finagle-chirper`). Unlike others, EMISSARY-L2 does not show any significant slowdowns.

Figure 3.7 also shows that EMISSARY-L2 policies outperform all of the others in terms of speedup and energy savings. The preferred configuration, P(8):S&E&R(1/32), performs consistently better than P(8):S&E. This is because the random filter tends to require lines to prove themselves with multiple starvations before being marked high-priority. This filters single reference lines very effectively, but it also filters single decode starvation lines just as well. This is important because high-priority reservations should be allocated to lines that starve with high probability and frequency, especially since an early single starvation is possible due to branch mispredictions and warm-up as new regions of code are executed.

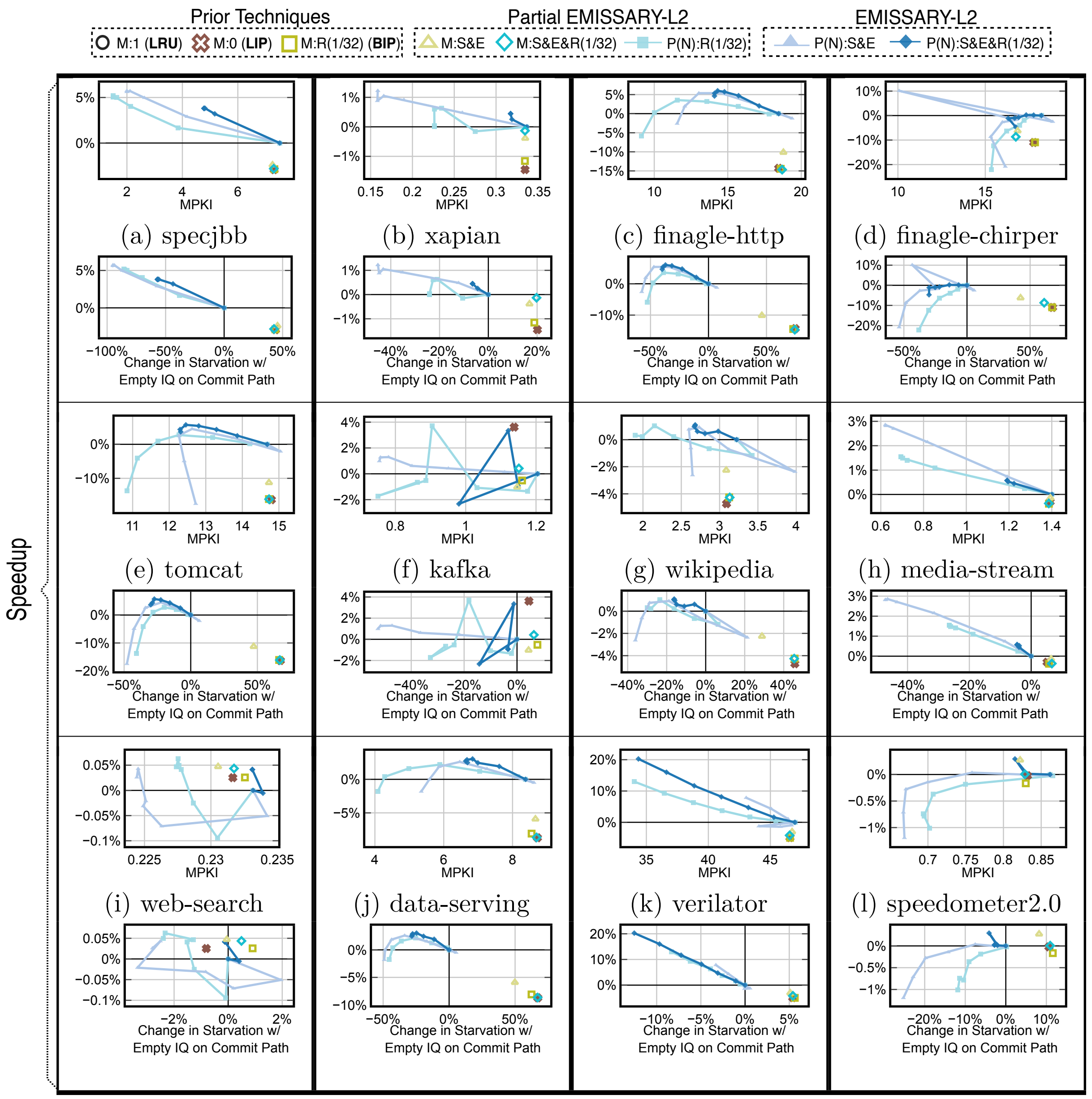


Figure 3.5: Speedup vs. L2 Instruction MPKI and Speedup vs. Change in Decode Starvation cycles when issue queue is empty for instructions on the committed path. P(N) techniques are shown as line segments with points corresponding to values of N from 0 to 14 in increments of 2. Lines connect P(N) to P($N+2$). TPLRU ($N = 0$) serves as the baseline.

Replacement policies such as SRRIP [43], BRRIP [43], and DRRIP [43] are designed to keep reused lines longer in the cache than the ones that are either used less frequently or have no reuse. Figure 3.3 shows that the hit rate at L2 is very high compared to the miss rate. In such a scenario, reused lines reach the highest priority state very quickly, and very often, this is the case at L2 in the datacenter workloads studied. When all ways in a cache set reach a high priority state, then the state of all lines is reset to a low priority state. In SRRIP policy, a newly inserted line would stay in the cache longer than in BRRIP policy, where only 3% of lines stay longer. BRRIP policy is detrimental when the newly inserted are evicted before they can be promoted to a higher priority state. A dynamic policy DRRIP is designed to reduce the negative effects of BRRIP and SRRIP. A dynamic policy dedicates a small sample (32 sets) to each policy and decides the winning policy based on the policy that contributes to fewer misses. Since the hit rate is much higher than the miss rate at L2, deciding the winner based on the miss rate is detrimental in the datacenter workloads studied. In a scenario where the L2 capacity is limited, the EMISSARY-L2 identifies a small fraction of long reuse instruction lines that caused starvations in the processor pipeline and preserves them in the L2 cache longer.

3.5.6 Contextualizing EMISSARY-L2's Benefits

EMISSARY-L2's impact is significant given how often the modeled architectures tolerate L1I misses [40]. Prior work suggests that increasing the front-end performance of a modern processor with a *properly tuned* FDIP front-end is extremely difficult [40, 41]. These works show that FDIP alone outperforms the latest stand-alone prefetching policies such as EIP (one of the top prefetchers in IPC-1) by 2.2% [41]. The authors further claim that a non-realizable `Perfect` prefetcher provides just 5.4% of the performance gains [40, 41, 69]. The `EIP-128KB` prefetcher improves FDIP performance by 4.3% [40, 41, 69]. It does this with a significant hardware storage cost of 128KB. In contrast, EMISSARY-L2 provides up to 3.24% with only 4KB of additional storage.

To further contextualize EMISSARY's performance, we compare EMISSARY-L2 to a perfect and unattainable model with zero cycle miss latency for all capacity and conflict instruction cache misses in the L2. The aforementioned zero cycle miss latency model achieves a geomean speedup of 15% over the FDIP baseline. EMISSARY-L2 achieves 21.6% of this unrealizable gain with only 4KB of additional state.

Finally, we also compare EMISSARY-L2 to DCLIP, DRRIP, and PDP. These techniques achieve geomean speedups of -2.48%, -2.9%, and -3.36%, respectively, when implemented on top of the FDIP baseline for the workloads studied in this work.

3.5.7 Persistence, By Itself, Improves Hit Rate

Figure 3.5 shows that, in a majority of the programs, to a point, as N increases, L2 Instruction MPKI proportionately reduces. In other words, EMISSARY-L2 techniques not only reduce starvation but MPKI as well. Even as the number of ways available to a significant fraction of low-priority addresses is reduced, misses decline as well. This was observed previously with the BIP technique [65] as well. With the prevalence of single reference (or extremely long time between reference) addresses, dedicating fewer cache resources to such lines makes way for lines that would otherwise miss. In this aspect, starvation acts as a filter, increasing the probability of isolating such lines by assigning them low-priority. Put another way, it helps reduce the extent to which these types of single reference lines can pollute the cache.

3.5.8 Impact on Back-end Stalls

EMISSARY-L2's impact on commit path front-end and back-end stall cycles is shown in Figure 3.6. Specifically, it depicts the reduction in stall cycles of the preferred P(8):S&E&R(1/32) policy when compared to the TPLRU baseline. Across benchmarks, EMISSARY-L2's impact on front-end stalls is more evident than its impact on back-end stalls. This is expected as EMISSARY-L2 is applied specifically to instruction lines. Inter-

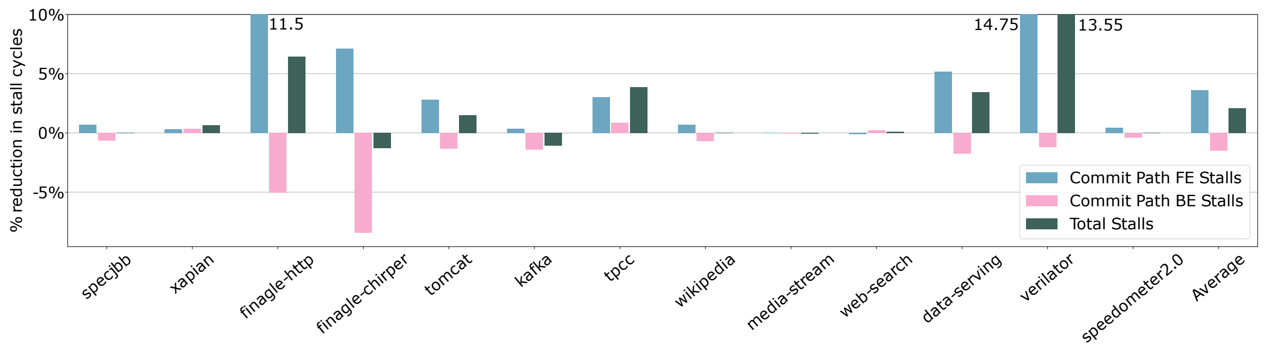


Figure 3.6: Reduction in various stall types of P(8):S&E&R(1/32) with respect to the TPLRU + FDIP baseline policy

estingly, many benchmarks show an increase in back-end stalls, but there is still an overall reduction in total stalls.

3.5.9 Energy Savings

We used McPAT [55] to model the energy consumption of different cache replacement policies explored. Fig. 3.7 shows energy savings for each benchmark and configuration. The energy savings are strongly correlated with the speedups achieved because of the relatively small amount of hardware added. In EMISSARY-L2 , there are only two bits added per cache line, one to mark the priority set once on insert and an additional one for TPLRU set on access. The EMISSARY-L2 P(8):S&E&R(1/32) configuration achieves a geomean reduction in the overall energy of 2.12% (up to 17.67%).

3.6 Balancing Data Lines

The EMISSARY-L2 policy advocated for in this work has $N = 8$ maximum ways reserved for high-priority instruction lines[3]. In a 16-way L2 cache, this policy reserves up to half of the ways for instructions. As mentioned in §3.1, once a cache with an EMISSARY-L2 policy reaches N high-priority lines in a set, the number of high-priority lines can never be reduced. In this section, we study the number of sets in the L2 cache that get saturated by

[3]Much of this content is common with [58], a work with co-authors

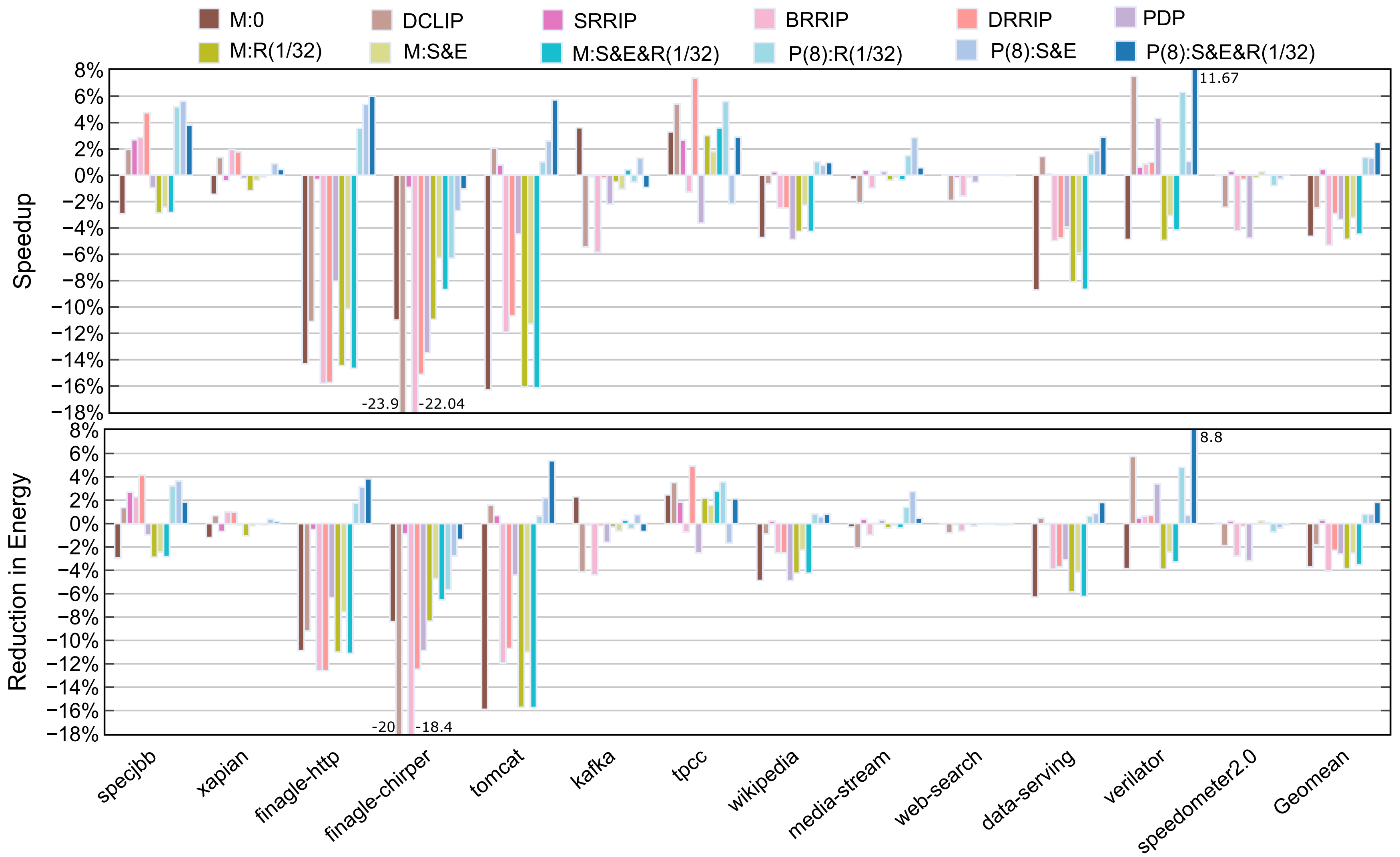


Figure 3.7: Speedup and Energy Reduction of a range of techniques relative to TPLRU + FDIP baseline policy

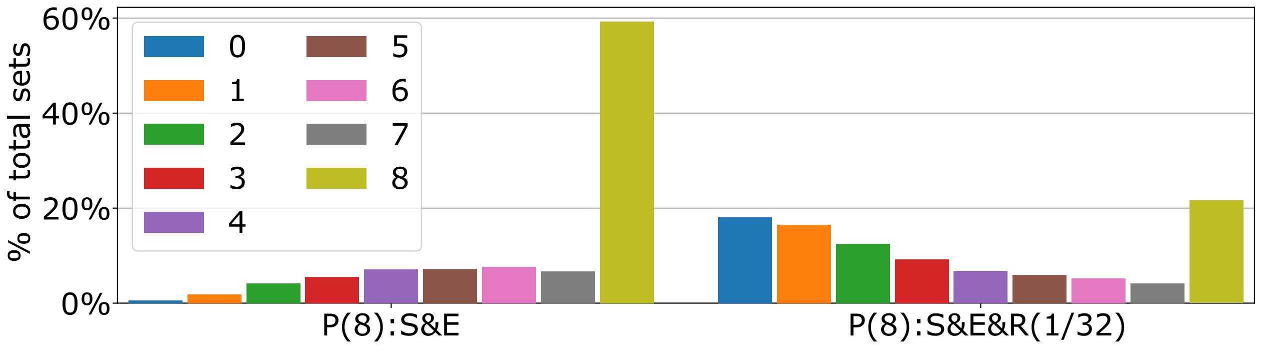


Figure 3.8: Distribution of high-priority lines across all sets when guided by P(8):S&E and P(8):S&E&R(1/32) policies, averaged across all benchmarks at the end of simulation.

high-priority instruction lines (i.e., 8 lines in a set are dedicated to instructions) and propose methods to minimize their impact on caching data lines.

Figure 3.8 shows the distribution of the number of cache lines occupied by high-priority lines when using the P(8):S&E and P(8):S&E&R (1/32) policies among all sets in the L2 cache at the end of the simulation averaged over all programs. With P(8):S&E, `finagle-chirper`, `tomcat`, `tpcc`, and `verilator` saturate (reaches N) for all sets. Less than 25% of all sets observe saturation with the highly selective (and more desirable) P(8):S&E&R(1/32) policy. In simulations of 1B instructions, resetting all $P = 1$ bits every 128M instructions has a negligible impact on performance.

Chapter 4

Criticality-Aware Instruction Prefetching

4.1 Introduction

Modern data center and cloud applications are becoming increasingly complex, featuring a code stack that spans several layers of software. As a result, these applications often exhibit instruction footprints much larger than the instruction cache, often even the L2 cache. Moreover, the trend is continuing toward even larger instruction footprints [45, 25]. Applications with such large code footprints are typically dominated by front-end bottlenecks, as shown in Figure 4.1, which analyzes one important, representative workload (top-down analysis [78] obtained on Alderlake desktop CPU using Intel's VTune profiler [7]). This shows three times as many issue slots lost to front-end bottlenecks than slots used for instructions that actually commit. Large code footprints put enormous strain on the instruction cache (L1I), with capacity misses inducing a large number of stalls [45] in the instruction fetch unit. This limits the number of useful instructions flowing into the pipeline backend. Increasing the cache size can address this problem, but at a large area and power cost, and creates implementation challenges related to meeting strict timing constraints, as the L1I sits on

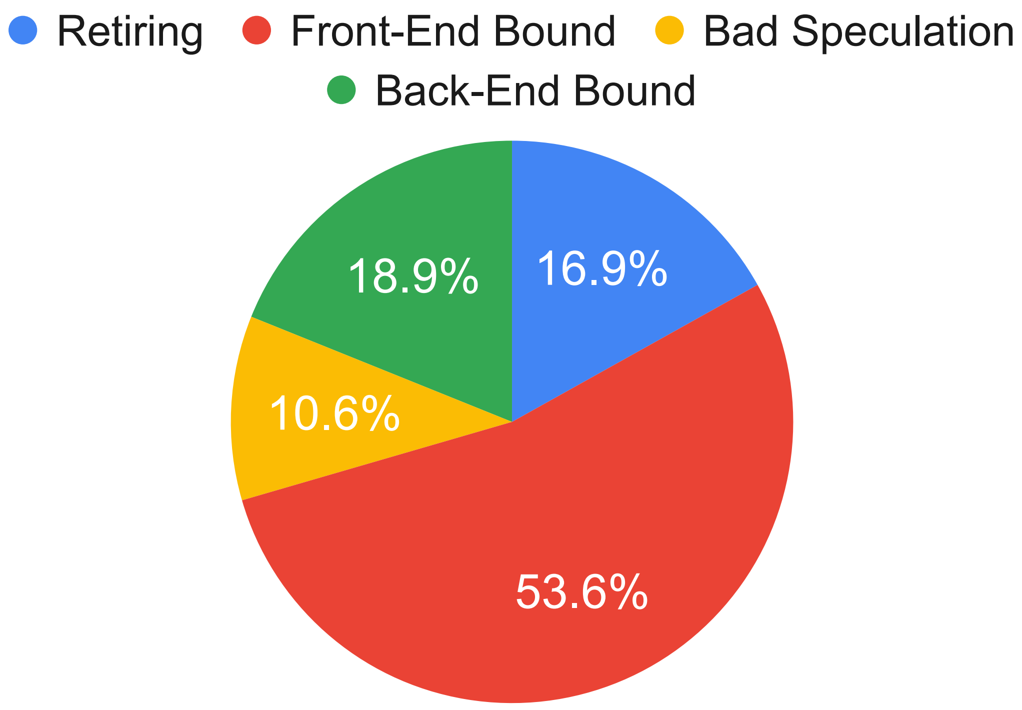


Figure 4.1: Top-down issue slots breakdown of cassandra benchmark on Alderlake host machine.

the critical path. Prefetching has the potential to address this bottleneck at a lower cost [33, 34, 25, 50, 56, 47, 73]. However, these techniques have been less effective for datacenter and cloud workloads that exhibit instruction footprints several orders of magnitude larger than traditional server applications [53, 48, 54]. Modern processors implement a decoupled front-end, aka fetch directed instruction prefetching (FDIP) [66], as an attempt to remedy this problem [62, 37, 72, 20]. With FDIP, the L1I fill is disassociated from its demand access, thus allowing the front-end to aggressively prefetch along the predicted path with very little sensitivity to decode and back-end back-pressure. This allows the FDIP prefetcher to hide most or all of the latency of L1I misses.

FDIP dramatically changes both the access/miss pattern seen by the L1I, and the criticality of misses (some misses are completely hidden by FDIP, others are not). Recent work on instruction prefetching [69, 38, 22, 59, 52, 35, 75, 23] have shown they can be effective in reducing misses, but either fail to account for the existence of FDIP or the variance in the criticality of those misses. This work, in contrast, seeks to augment FDIP with a new instruction prefetcher that focuses on only those misses whose latency is not already tolerated by FDIP. The defining characteristic of those misses is often their distance from a branch mispredict (or other front-end mispredict/hazard). Misses far removed from a mispredict are typically fully covered by the FDIP prefetch with its ability to run far ahead. Misses that occur shortly after mispredicted branches are typically not hidden. This chapter presents Priority Directed Instruction Prefetching (PDIP), a novel instruction prefetching technique that complements FDIP, only issuing prefetches for targets known to be *front-end critical*, or *FEC*; that is, misses that in the past have truly resulted in front-end stalls because they were insufficiently hidden by FDIP. Prior work (EMISSARY-L2 [58]) showed that front-end criticality could be used to design a more effective instruction cache replacement algorithm. In this work, we also show that even in the presence of a FEC-based replacement policy, many FEC misses remain. Thus, an FEC-based prefetch mechanism can augment, and even be synergistic with, an FEC-based cache. In PDIP, cache lines are marked FEC if a prior miss occurred along a resteered (i.e., after branch misprediction) predicted path, and exposed the front-end to one or more stalls. PDIP only considers those lines as prefetch candidates.

In addition, we need a trigger to initiate prefetches. This work shows that we can associate FEC cache lines with an instruction that caused a disruption of the front-end (since it requires a disruption to empty or stall the Fetch Target Queue, preventing FDIP from hiding the latency). Thus, PDIP achieves timely prefetch by triggering the prefetch of FEC cache lines when it sees the associated instruction. In summary, this work makes the following contributions:

- We describe the design of PDIP and show how it addresses two key issues impeding instruction prefetching today, namely low prefetch effectiveness and high storage requirements.

- We present an evaluation of PDIP alongside a best-effort evaluation of EIP (Entangled Instruction Prefetcher) [69]. To the best of our knowledge, EIP is the first instruction prefetching work to consider FDIP. Using a series of 16 large footprint workloads on a detailed processor simulator modeled after a Golden Cove machine [17], we demonstrate a geomean IPC gain of 3.2% across all benchmarks for only 43.5 KB storage cost for PDIP, against a gain of 1.5% for EIP at similar hardware budget.

- We show that even in the presence of a front-end criticality based cache replacement algorithm such as EMISSARY-L2 [58], PDIP is still able to provide great value, realizing a geomean IPC of 3.7% across all benchmarks.

4.2 Background

This section provides background knowledge of decoupled front-end microarchitectures and their implications on instruction prefetching techniques. It also describes prior work [58] on front-end criticality-aware cache replacement.

4.2.1 Decoupled Front-end

Figure 4.2 shows a decoupled front-end machine where the instruction fetch unit (IFU) is decoupled from the instruction address generator (IAG) via the Fetch Target Queue (FTQ). The branch prediction unit (BPU), a part of the IAG, includes the conditional branch predictor, the direct jump address predictor (aka BTB), the indirect jump predictor, and a return address stack, all feeding the IAG to speculatively compute the address of the next instruction block to be fetched. The FTQ is a FIFO queue that is filled with the targets computed by the IAG along the predicted path. The cache lines en-queued in the FTQ are prefetched into the L1I. Thus, non-resident instruction blocks can be prefetched into the L1I when the address enters the FTQ rather than on demand when the address reaches the IFU.

It is common for the IAG to exceed the throughput of the backend, keeping the FTQ full in the absence of squashes in the pipeline.

An important component in a decoupled front-end machine is the depth of the FTQ, which determines how far the predicted instruction stream (which drives the prefetching) can get ahead of the actual fetch demand. A sufficiently large FTQ offers a deep enough instruction prefetching window such that a full L1I miss, and possibly even an L2 miss, can be tolerated without stalling the IFU. This assumes the FTQ is full, given that the BPU nearly always sustains a higher throughput than the backend, the FTQ is generally full in the absence of FTQ resetting events (e.g., branch mispredicts). Modern decoupled front-end processors, therefore, implement deep FTQs to get the most benefit from prefetching and tolerate cache miss latency. The importance of FTQ in the context of instruction prefetching was amply discussed in [40]. The authors show that most of the performance benefits obtained with recent instruction prefetching proposals [69] vanish with a decoupled front-end machine implementing a modest 24-entry FTQ. The key insight here is that in the presence of FDIP, which has the potential to hide the full latency of the majority of misses, we see high variance in the performance-criticality of L1I misses. Some misses have no impact on front-end (or, naturally, backend) performance, while others still do. In this work, therefore, we show that limited prefetching resources should be focused on only the latter, the front-end critical (FEC) misses. A line is considered as FEC when it meets the following conditions: (1) the line must have retired an instruction, (2) the line must have missed the instruction cache, and (3) the line must have produced front-end stalls as a result of the miss.

4.2.2 Front-end Critical Cache Replacement

While this work is the first to consider front-end criticality in the prefetcher, in this section, we describe work that accounts for FEC misses in the cache itself.

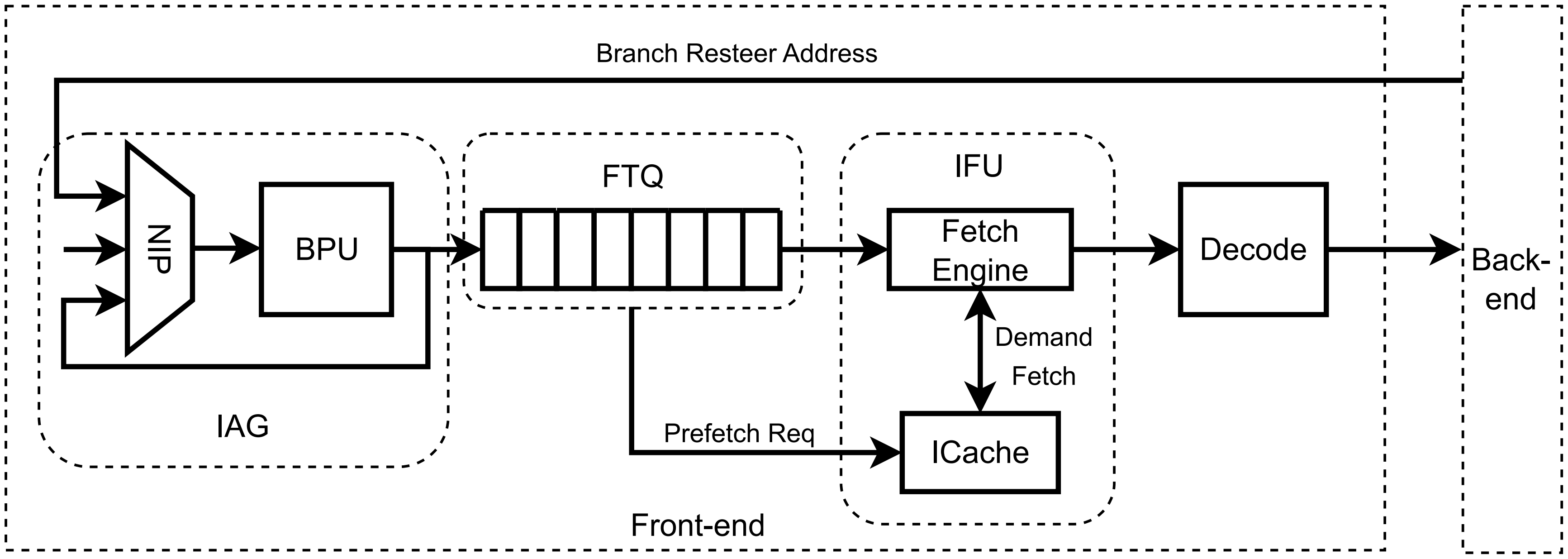


Figure 4.2: Generic decoupled front-end microarchitecture

The EMISSARY-L2 [58] cache replacement policy identifies lines as front-end critical, and gives such lines priority in the replacement policy.

The main idea behind EMISSARY-L2 is to preserve the FEC instruction lines in the L2 cache. Lines identified as priority L2 cache lines in that work (which we call FEC lines in this chapter) are given higher priority for retention at eviction over the lines that are not preserved (non-priority lines). Each cache block has an additional status bit called P-bit (Priority).

Information pertaining to front-end stall exposure and cache miss are collected at decode time and cache access time, respectively, and then propagated to the retirement stage. Then, when a line retires an instruction, these conditions are checked, and the decision to label the line is made.

The EMISSARY-L2 policy shields up to n ways per set. They show that protecting up to 8 ways shows the best performance, which is also confirmed by our exploration. FEC lines are promoted in an EMISSARY-L2 cache by setting their corresponding P-bit. To avoid over-promotion of FEC lines, only 3.125% of all FEC lines are promoted. This helps it avoid over-reacting to single-instance FEC lines.

In this paper, we show that PDIP is synergistic with EMISSARY-L2 in two ways. First, there is a physical synergy – we only need one mechanism to identify FEC misses, and we can exploit it either in the cache [58] or the prefetcher (this work), or both. Second, the two techniques do not redundantly attack the same problem, but rather are complementary – PDIP provides higher gains in a system with FEC-aware caches than in a system without.

4.3 Do We Need Another Prefetcher?

EMISSARY-L2 seeks to remove the most damaging (i.e., front-end critical) misses from the front-end, and improves overall performance as a result. Figure 4.3 shows improved front-end performance with an EMISSARY-L2 cache on top of FDIP, which exceeds the gains from

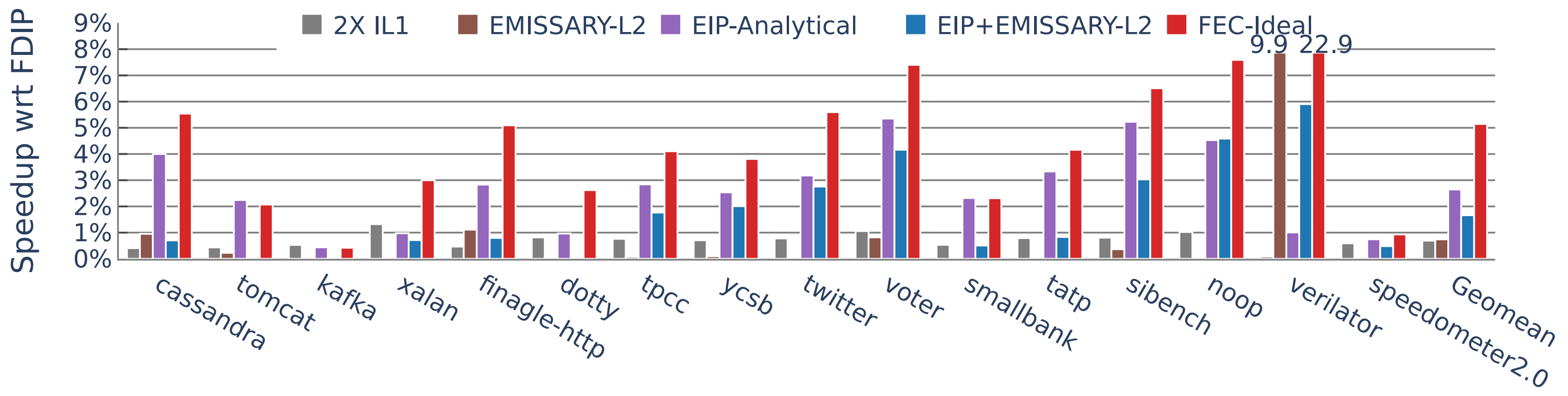


Figure 4.3: Performance gain of various prior techniques on all benchmarks

doubling the size of the L1I. However, this performance still falls far short of FEC-Ideal, which represents a front-end where every FEC line (as defined in the previous section) is fully hidden. This argues that there is still much to be gained by better handling these misses, and there is likely a role for an additional prefetch engine.

Also included in Figure 4.3 is the current state-of-the-art front-end prefetcher, EIP. In this figure, EIP-Analytical represents the performance-oriented version of the Entangling Instruction Prefetcher (EIP [69]) with a very large entanglement table (>200KB) that consumes six times the resources of the L1-I. FEC-Ideal refers to a system with an EMISSARY-L2 cache at L2, but where EMISSARY-marked FEC lines are always delivered with the fast latency of the L1I cache. EMISSARY-L2 policy is the EMISSARY-L2 cache at L2 with 8 protected ways per set and 2X IL1 is the configuration with L1I twice (64KB) the size of the baseline configuration (32KB). As shown in the figure, EIP-Analytical outperforms EMISSARY-L2 but still falls short of FEC-Ideal. Furthermore, when EIP and EMISSARY-L2 are combined, they can end up hurting performance if they are not designed to complement each other. This not only underscores the need for a better prefetcher but one that works in conjunction with a FEC cache replacement policy such as EMISSARY-L2 .

Figure 4.4 shows FEC lines of each benchmark as a percentage of all instruction lines accessed in the retired path (first bar) and decode starvation cycles caused by FEC lines compared to total decode starvation cycles (second bar). It shows that only 10% of lines (the FEC lines) are causing 62% of decode starvation cycles. FEC lines that cause more than ten cycles of decode starvation we call *high-cost FEC lines*, which constitute 5.08% of all lines. Most of those (4.26% of the total) also result in an issue queue empty signal, which means the back-end is also stalling. High-cost FEC lines contribute to 56.15% of all decode starvation cycles. High-cost FEC lines with back-end stalling contributes to 46.83% of decode starvation cycles. This demonstrates we can focus our prefetcher on a very small subset of fetched lines but still achieve most of the available gain.

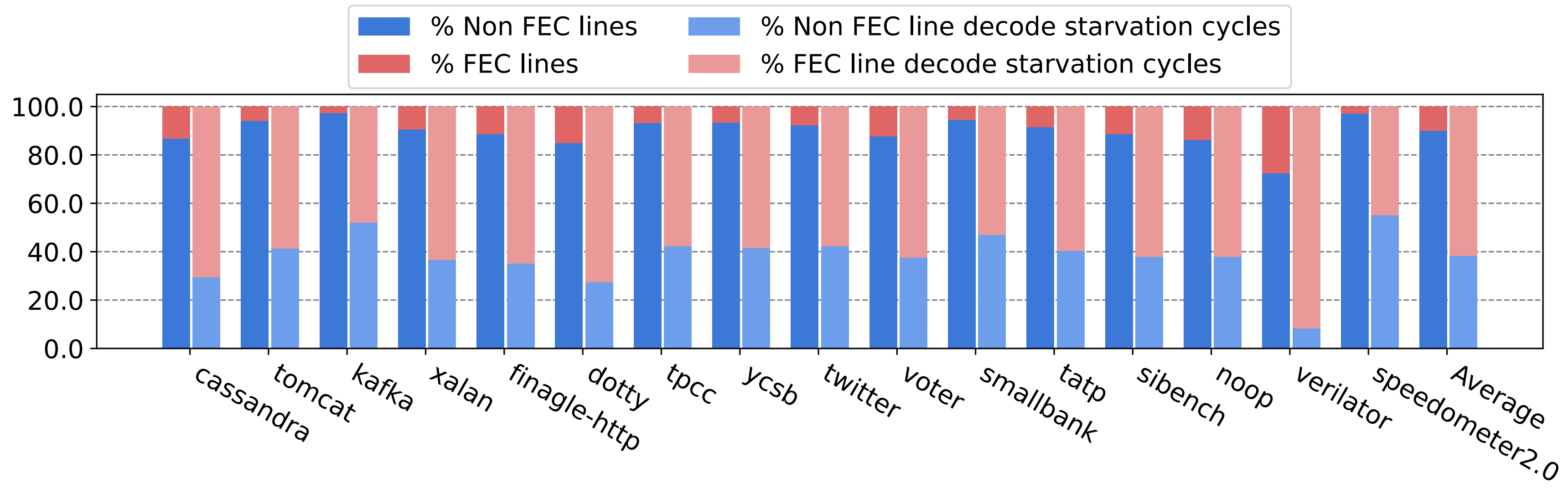


Figure 4.4: First bar shows the dynamic number of FEC lines as percentage of total lines. Second bar shows the decode starvation cycles caused by FEC w.r.t total decode starvation cycles.

4.4 PDIP

In designing a prefetcher for an FDIP-equipped processor, it is important to understand that FDIP already represents a highly effective prefetcher. Thus, any new prefetcher must be carefully designed to complement FDIP rather than run independently. This section describes PDIP, our Priority Directed Instruction Prefetcher, which identifies specific instances where FDIP's effectiveness at prefetching instructions is impaired, such as on the recovery path of a mispredicted branch, as illustrated in Figure 4.5.

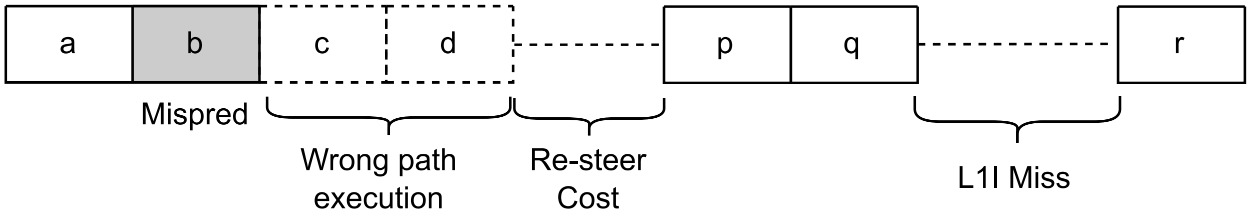


Figure 4.5: An example showing a sequence of instructions. Each box shown represents one instruction. Dashed boxes are the instructions in the wrong path and dashed line shows wasted cycles due to resteering along with the cost of a miss in instruction cache

In the case of a mispredicted branch, the front-end pipeline is flushed, including the FTQ; thus, the latency of cache misses on the resteer path of a branch is exposed and cannot be tolerated by FDIP because they appear in the empty or near-empty FTQ with insufficient lead time to prefetch them effectively. PDIP attempts to select both the right prefetch candidate, e.g., block *r* in the figure, and identify the appropriate trigger instruction for the prefetch candidate, e.g., block *b* in the figure. In this way, PDIP essentially jump-starts the FTQ, prefetching instruction blocks before their addresses appear in the FTQ, but only for those blocks whose addresses will appear in the FTQ; it is too late to prefetch effectively.

4.4.1 Selecting Prefetch Candidates

Drawing from the insight that the performance criticality of the instruction cache misses is highly variable, PDIP only considers prefetch candidates among lines identified as high-cost FEC line, which also experienced back-end stalls. This serves two purposes. First, because most instruction cache lines never reach FEC status, the number of unique prefetch candidates to track is, in most cases, limited, heavily reducing storage cost requirements compared to prior works [69, 22, 23, 75]. Second, by considering prefetch candidates only among FEC lines, PDIP filters out a significant number of unnecessary prefetches, improving its effectiveness.

4.4.2 Selecting a Trigger Instruction

We will use the example in Figure 4.6 to illustrate how PDIP finds a trigger and associates it with a prefetch candidate.

As discussed in the previous section, PDIP selects its prefetch candidate exclusively among FEC lines. Based on the conditions for promoting a line to FEC status (same as [58]), the line must have met three conditions. It is exposed to front-end stalls after missing in the instruction cache and the line must have retired at least one instruction (the line was not fetched on the wrong path). In the figure, for example, when the block labeled r retires the first time (see Retire column in First Instance), it has missed in the cache after it has been exposed to pipeline bubbles. Block r is therefore considered by PDIP as a prefetch candidate.

Because FDIP-fetched lines should be fully hidden in a full, large FTQ, and because the FTQ should be full when execution is sufficiently removed from a resteer event (a *resteer* event is one that resets and redirects the front-end, emptying the FTQ), this implies that a miss that incurs front-end stalls has been fetched in the wake of a resteer event. In our example, block r in the figure sits on the resteer path of the instruction that caused the FTQ to be flushed – block b, which was mispredicted and caused a pipeline flush.

PDIP, therefore, associates the trigger instruction of the prefetch candidate, i.e., block r in the figure, to the front-end resteering instruction, i.e., block b. PDIP picks the front-end resteering instruction according to the type of front-end stall event. There are a couple of categories of events that can expose L1I misses to front-end stalls. The first are those that

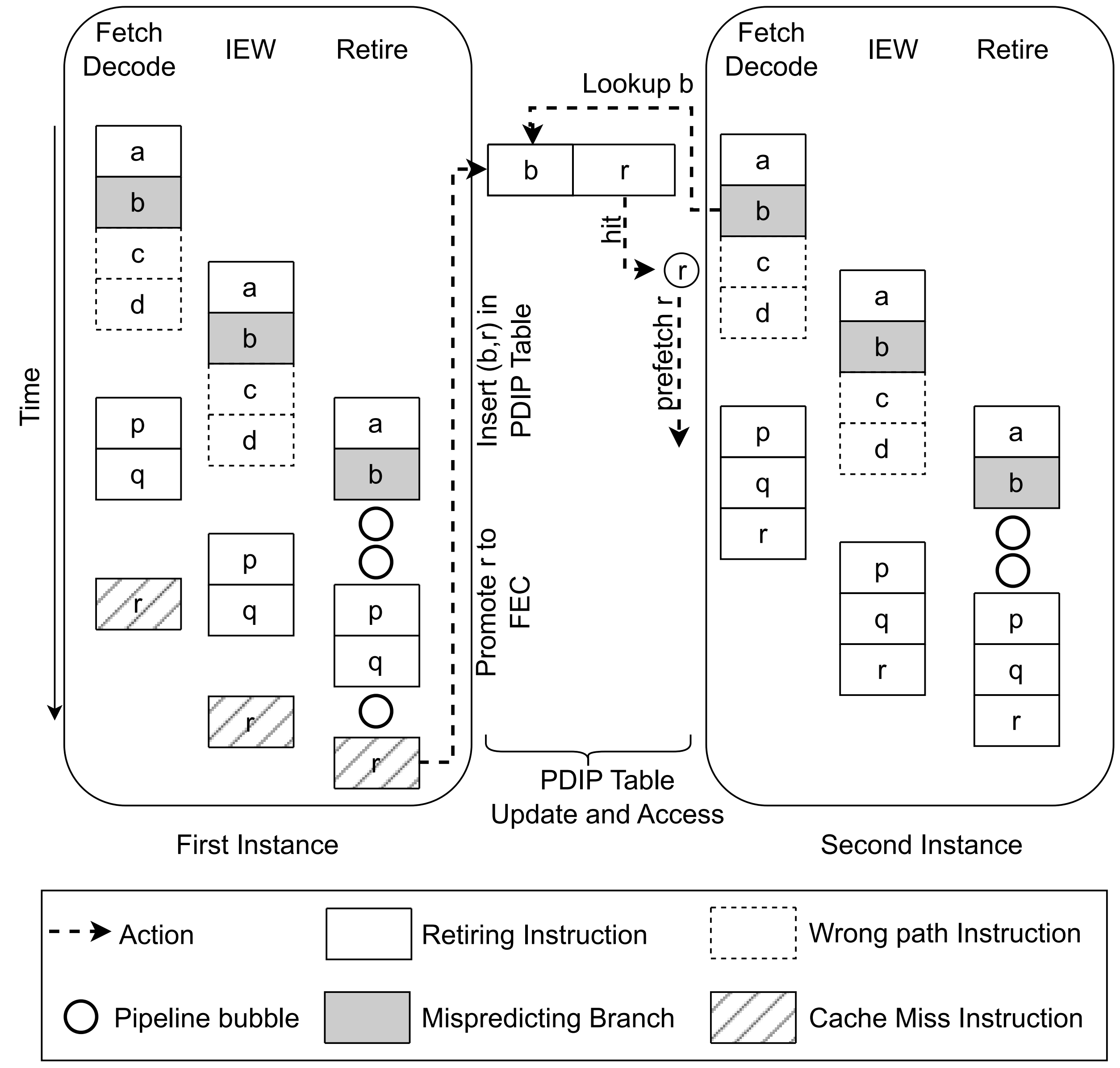


Figure 4.6: An example showing sequence of instructions in the processor pipeline.

cause a resteer (flush) of the front-end. These include branch mispredicts (including BTB target mispredicts) and BTB misses. These each expose the front-end to stalls because when the FTQ is empty, the interval between prefetch (FTQ entry) and demand fetch will be too short. The other category is latencies that exceed the ability of the FTQ to hide. This includes L1I misses that miss in both the L1I and the L2 (and possibly L3 as well). These will also be marked as FEC because they incur front-end stalls even in the presence of a full FTQ. It should be noted that we also experimented with instruction TLB misses as a trackable event that can also expose the front-end to cache-miss-related stalls, but saw no performance gain in doing so, so these results are not included. But it is possible that other workloads would be sensitive to those.

For front-end stalls due to control flow mispredicts, PDIP identifies the trigger instruction as the mispredicting branch instruction, e.g., block b in the figure, or the instruction missing in the BTB. For front-end stalls in the absence of a resteer event (i.e., long-latency misses) PDIP identifies the trigger instruction as the last taken branch instruction that was retired.

The trigger instruction (block b in our running example) and its prefetch candidate (block r) are tracked by means of a PDIP table. The table is accessed once per new FTQ entry since an FTQ entry represents a basic block (see block b under the Fetch column in Second Instance). On a match, a prefetch to the associated target address is issued, e.g., prefetch r in the figure, which eliminates the bubble observed previously, as shown in the figure with the block r retiring without encountering a bubble in the Second Instance.

4.4.3 Synergy between PDIP and FDIP

FDIP hinges on the ability of the BPU to predict a stream of instruction addresses that the IFU can then prefetch into the instruction cache. The control flow prediction structures in a BPU, namely the BTB and the various history tables, are therefore critical to FDIP efficiency since they all contribute in some form to the BPU accuracy. The BTB keeps track of taken branches and provides a target prediction, while the history tables are used to produce a direction prediction for these branches. FDIP outperforms other prefetch mechanisms because the stream of predicted instruction addresses is the highest quality prediction of future L1I accesses available.

Data center and cloud workloads put enormous strain on these control flow prediction structures that are not sufficiently provisioned to handle their large code footprints. The BTB, for instance, is not large enough to track all taken branches, and the history tables are not big enough to capture enough state to produce a prediction for each such branch with high confidence. For FDIP, this translates to reduced opportunities to prefetch down the BPU predicted path since these branch mispredicts cause a pipeline flush of front-end structures.

Worse, these large-footprint applications also place tremendous pressure on the L1I. Thus, as FDIP loses effectiveness due to the increased pressure on the branch predictor, the problem that FDIP is trying to solve (L1I misses) is also exacerbated. Put another way, with FDIP, the cost of a mispredict is threefold – the traditional two costs (the cost of squashing the wrong path and the cost of refilling the pipeline) plus a new one, the exposure to front-end L1I misses that FDIP can no longer tolerate. And as the code footprint continues to increase, the third cost can begin to dominate.

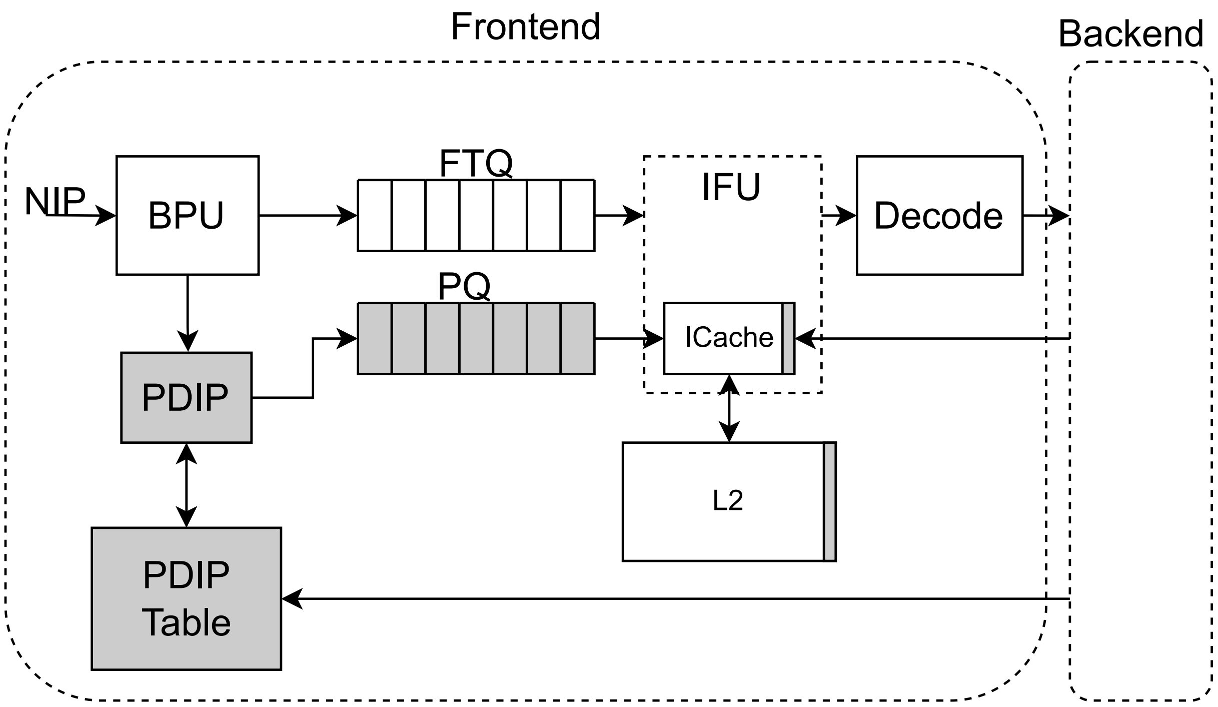


Figure 4.7: PDIP Pipeline showing new components added in gray blocks

4.5 Design Implementation

Figure 4.7 shows the main PDIP building blocks and how they integrate into a decoupled front-end processor. The BPU feeds the PDIP controller with the block address of a branch on a BTB hit or the block address of the Next Instruction Pointer (NIP) on a miss. The PDIP controller uses this address to index the PDIP Table to retrieve the address of a prefetch candidate. This address is expanded to a full physical address and sent to a Prefetch Queue (PQ). PQ enqueues the address only if there is a free entry and enough MSHR registers to handle demand requests; otherwise, the address is dropped. This is done to ensure demand requests are not penalized by aggressive prefetching. PQ probes the instruction cache with each entry and only sends a prefetch request to the next level cache on a probe miss and if there are still enough MSHR registers available; otherwise, the request is also dropped. A threshold of 2 entries is used to ensure demand accesses are not penalized. We empirically determined this value works best for our workloads.

4.5.1 PDIP Table

The PDIP table associates a prefetch candidate with a front-end stall-causing instruction, i.e., the trigger instruction. When the front-end stall-causing event is caused by a control flow hazard, the trigger is always a branch instruction. In the case of a long-latency miss, we choose the last taken branch as the trigger. In practice, however, we associate the prefetch candidate with the block address of the trigger instruction instead of the PC address of the trigger. Table insertion or look-up, therefore, uses a block address. This allows the PDIP Table to still be able to retrieve entries that miss in the BTB.

TAG	LRU	FEC Line 1	Mask				FEC Line 2	Mask			
b	1	r	1	1	0	0		0	0	0	0

Figure 4.8: A PDIP Table with two targets per entry

Because a block may contain more than one branch, it is possible that more than one prefetch candidate also maps to the same entry in the table. Thus, each entry in the table contains multiple prefetch targets. Each target can also indicate any of the following four cache blocks in the address space for prefetching via a 4-bit mask when they share the same trigger. This provides compaction and nicely handles basic blocks that span multiple cache lines. We show the design of the PDIP Table in Figure 4.8. A set associative table design is used to reduce conflict misses. All configurations of the PDIP table we evaluate use fixed 512 sets, and we vary the associativity appropriately. We validated that using a 10-bit tag reduces aliasing considerably.

The FEC line address field stores the physical address of a prefetch candidate. Mask bits in the example represent the third and fourth following blocks; thus, when triggered, block r, r+3*blocksize, and r+4*blocksize would be prefetched.

4.5.2 Optimizing Table Size

An indirect branch could potentially lead to a different target each time it executes. Similarly, a return instruction could jump to a different address each time the same function is invoked in a different calling context. We choose to ignore return jumps to reduce pollution in the table, but other indirect branches are inserted.

To improve prefetch accuracy, in addition to the tag, we also experimented with augmenting the table with path information of the last three branches leading to the trigger. A prefetch candidate is fed to the PQ only if both the TAG and path information match. The performance gains obtained (not reproduced here) were not significant enough to justify the added complexity of the design.

4.5.3 Optimizing Table Occupancy

Even focusing the prefetcher only on FEC lines, we still found some cases with significant cache pollution. Thus, we examined two mechanisms to reduce pollution with minimal impact on the most effective prefetches, and both work by inserting lines more selectively into the PDIP table. First, we insert only high-cost FEC lines that cause back-end stalls in the table. Second, we insert into the table with a reduced probability – in this way, a line marked FEC once is less likely to be inserted, but any line repeatedly marked will be inserted. We examined probabilities from 1 to .03 and found .25 to provide the best overall gains.

4.5.4 Hardware Storage Overhead

The Metadata of PDIP involves two components: an augmented cache to track FEC lines and the PDIP table, which stores the triggers and targets. The storage overhead of the bit to identify FEC lines (included in EMISSARY-L2 , but also used by PDIP) in both the L1I and L2 would be about 4 KB for our configuration. Our default PDIP table configuration has 512 sets, an 8-way set associativity with two physical address targets, and a 4-bit offset mask per entry. Table sizes are scaled by increasing associativity. Each target requires 34 bits of physical address, each tag is 10 bits, one LRU bit is used for each way, and 4 bits of mask per target, which results in 356352 bits (43.5KB) of storage for a table with 512 sets and eight ways.

4.6 Simulation Methodology

This section provides a description of the simulation infrastructure, the large code footprint workloads, and policies used to evaluate PDIP.

4.6.1 Simulation Model

Our baseline CPU configuration is modeled after a Golden Cove [17] (commercially known as Alder Lake) CPU core microarchitecture using the gem5 [27] simulator. Table 4.1 shows some of the key parameters modeled in this study. Workloads are simulated using an out-of-order, execution-driven CPU model (O3CPU) in Full system simulation, which models a full operating system (Ubuntu) and are running multi-threaded JAVA applications. The O3CPU models the wrong path execution.

The workloads are first warmed up for approximately 10 million instructions, during which time the caches, branch predictor, and other structures are also warmed up. After this warm-up period, the simulation switches to detail mode (O3CPU) and runs for a further 100 million instructions.

4.6.2 Baseline Description

A key contribution and distinguisher of this work is the fact that we faithfully model a very aggressive processor front-end, extending gem5's O3CPU model to implement Fetch Directed Instruction Prefetching (FDIP), thus supporting a decoupled front-end. Since performance of FDIP directly correlates with the accuracy of the branch predictor, we made improvements to the BPU of gem5 by adding an ITTAGE indirect predictor [44], using a large BTB (8K Entries) and fixed several bugs. We have added support to the BPU indirect predictor and the BTB to enqueue the predicted cache lines into the FTQ. The FTQ can directly issue prefetches in the L1I. We also model a prefetch queue (PQ) alongside the FTQ to support various prefetch policies explored in this paper. To ensure we don't have duplicate prefetches, targets are checked against the FTQ before issuing a prefetch. Control flow resteers would flush the FTQ before resuming fetch from the correct path. As gem5 is execution-driven, the wrong path effects of such resteers are also accurately modeled. We have also added an early correction feature in the front-end where a branch PC is pre-decoded at the time of fetch to identify bogus branches in the BTB and resteer the FDIP pipeline. Our updated FDIP model provides a 27.1% improvement over the standard O3CPU model without FDIP. We use a 24-entry FTQ in our baseline (each entry represents a basic block), which strikes a balance between being deep enough to tolerate miss latency while preventing the front-end from being overly aggressive and introducing negative effects. FDIP is a structural change to the front-end pipeline of the processor and has been a key feature in the industry for over a decade. Thus, we believe any front-end CPU microarchitecture work has little relevance without FDIP and should build upon this baseline with FDIP. We utilized our FDIP-supported gem5 as the baseline for all experiments presented in this study – thus addressing the concerns raised by Ishii et al. [40] on the need for a representative baseline in academia.

Field \ Model	**Alderlake like**
ISA	X86
Private L1-I Cache	32kB (8-way, 64B) 2 cycle hit, 16 MSHR
Private L1-D Cache	64kB (16-way, 64B) 2 cycle hit, 16 MSHR
Private L2 Cache	1MB (16-way, 64B) 10 cycle hit, 32 MSHR
Shared L3 Cache	2MB (16-way, 64B) 20 cycle hit, 64 MSHR
Branch Predictor	TAGE (64KB)[76]/ ITTAGE(64KB)[74]
BTB size	8K entries (119.01 KB)
FTQ	24 entry [40]
Prefetch Queue	40 cachelines
Decode/Retire	12 wide
ROB Entries	512
Issue/Load/Store Queue	194/ 144 / 112
Int/Vec Registers	448 / 400

Table 4.1: Processor configurations

4.6.3 Benchmarks

We use 16 widely used client-side and server-side multi-threaded workloads with large code footprints to evaluate PDIP. Table 4.2 contains all front-end heavy benchmarks used from various benchmark suites [28, 64, 29, 21, 5]. Benchmarks with an L1I MPKI of over 20 are used in this work. We validated characteristics of these workloads by Top Down analysis using Intel's VTune on a Linux System with Alderlake CPU.

Benchmark Suite	**Benchmarks**
DaCapo [28]	`cassandra` [1], `tomcat` [4], `kafka` [2], `xalan`
Renaissance [64]	`finagle-http` [15] , `dotty` [6]
OLTB Bench [29] (PostgreSQL [9])	`tpcc` [14] , `ycsb` [18], `twitter`, `voter`, `smallbank`, `tatp`, `sibench`, `noop`
Chipyard [21]	`verilator` [16]
Browser Bench [5]	`speedometer2.0` [13]

Table 4.2: Benchmarks used to evaluate PDIP.

4.6.4 OS and IO bottlenecks : Full System

In a Full System simulation, OS and IO bottlenecks could impact the overall performance. Thus, we spent significant time minimizing noise from OS (scheduler interrupt) and IO (disk interrupts) to ensure negligible (on average less than 0.2%) divergence in OS effects between runs. For example, one trick we deploy is to use "retired instruction counts" rather than cycle counts to drive the OS scheduler quantum, thus producing far more repeatable runs even when the optimizations produced different timing characteristics. The OS scheduler decides which process thread to schedule on available CPUs. To reduce noise, we use a real-time scheduling policy for the benchmarks with 100% of CPU time dedicated to real-time processes. Another cause of divergence is due to IO events. A process waiting on IO is usually switched out to allow other processes to use CPU resources. In order to reduce the variability introduced by IO events, we used very low latency for disk IO operations so that a process would not need to switch out while waiting for the disk. Even after using these and a few other tricks, we still observe negligible divergence caused by pending instructions in the CPU pipeline from the time an interrupt is received.

4.6.5 Policies Evaluated

EMISSARY-L2 offers two main configuration knobs, namely the number of FEC ways per set and a random probability to actually promote lines identified as FEC. We empirically found promoting FEC-qualified lines at retirement with a random probability of 1/32 and reserving eight ways in the L2 provides the best performance across our benchmark suite. Unless otherwise stated, then, we assume that FEC-qualified lines are promoted at retirement with a 3.125% probability, and the L2 cache preserves eight ways for FEC lines.

We evaluate PDIP alongside two other configurations. The first one, 2X IL1, is similar to our baseline but with twice the size of the L1I. The main idea with 2X IL1 is understanding the tradeoffs between increasing cache size vs. investment in new approaches. The second configuration is EIP [69], a recent instruction prefetching mechanism. That work evaluated EIP using the ChampSim [19] simulator. In this work, we model 2 versions of EIP in gem5, which enables a more faithful model of FDIP and, unlike ChampSim, accurately models wrong path execution. EIP-analytical is an analytical model that relaxes practical considerations. For instance, we assume accessing the entangling table and prefetching the entire basic block for each dst_entangled entry is done in one cycle. We model a history buffer with 40 entries and an unlimited entangling table. A history buffer size of 1024 was considered, but it did not provide any improvement over 40 entries. Thus, we used a 40-entry history buffer for all configurations of EIP. The entangling table is updated in the commit stage of the pipeline to avoid wrong path accesses polluting the table. The history buffer is implemented in the commit stage to contain only those entries in the correct path. L1I miss latencies are captured at fetch but used at commit to compute entangling distances. To avoid misses in the instruction TLB, the full physical address is stored in the entangling table. Similar to the PDIP pipeline, if the PQ is full, then any new prefetch request will be dropped. We also implement other prefetchers, EIP(S), that have stricter storage budgets (i.e., S KB).

The different policies evaluated are summarized in Table 4.3. We measure performance relative to baseline in Instructions Per Cycle (IPC), and we use Geometric Mean for the mean IPC speedup.

Policy Name	**Description**
Baseline	Golden Cove like core
EMISSARY-L2	Priority Ways{L2(8-ways)}
PDIP(S)	PDIP with S KB PDIP Table
EIP-Analytical	Analytical model of EIP [69] with large storage budget for performance
EIP(S)	EIP prefetcher with S KB storage
2X IL1	64KB Instruction Cache

Table 4.3: Policies Table

4.7 Evaluation

We evaluated the benchmarks discussed in Section 4.6.3 on the policies described in Table 4.3 using the following metrics: IPC, prefetch accuracy, prefetch coverage, and prefetch rate per kilo instructions. We examine multiple PDIP Table sizes.

Figure 4.9 provides data on our benchmark set, showing absolute MPKI when running on our baseline configuration. Average MPKI on the instruction cache, L2 instruction-side and L3 are very high, about 85.9, 12.4 and 3.06, respectively.

4.7.1 Performance Analysis

Figure 4.10 shows relative IPC gains across our benchmarks for the policies in Table 4.3. A PDIP Table of 512 sets and 8-way associativity is used in PDIP(44). In most benchmarks, PDIP(44) matches or outperforms EIP-Analytical while maintaining practical implementation considerations and utilizing five times less storage. The PDIP(44) shows a geomean speedup of 3.15% over the FDIP baseline as compared to a 1.5% speedup of EIP(46) at a similar storage budget.

As shown in Figure 4.3, EIP suffers when paired with EMISSARY-L2 , lacking synergy with the state-of-the-art replacement algorithm. Conversely, PDIP is carefully designed to complement both FDIP and EMISSARY-L2 and provides additional gains over each,

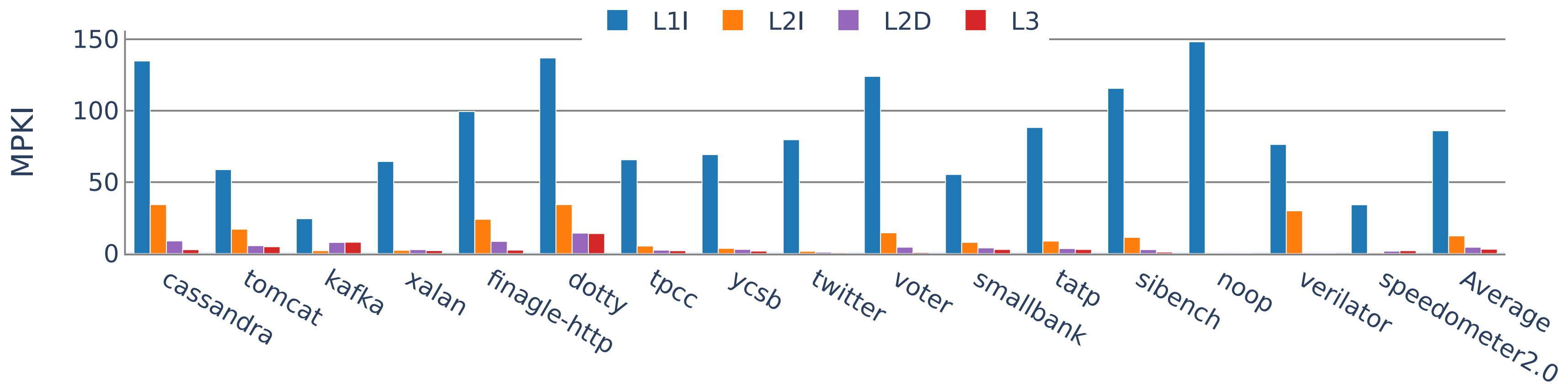


Figure 4.9: Misses Per Kilo Instructions (MPKI) at L1-I, L2-I and L2-D (instruction and data misses in the L2 cache, respectively), and L3 caches of benchmarks presented in this work

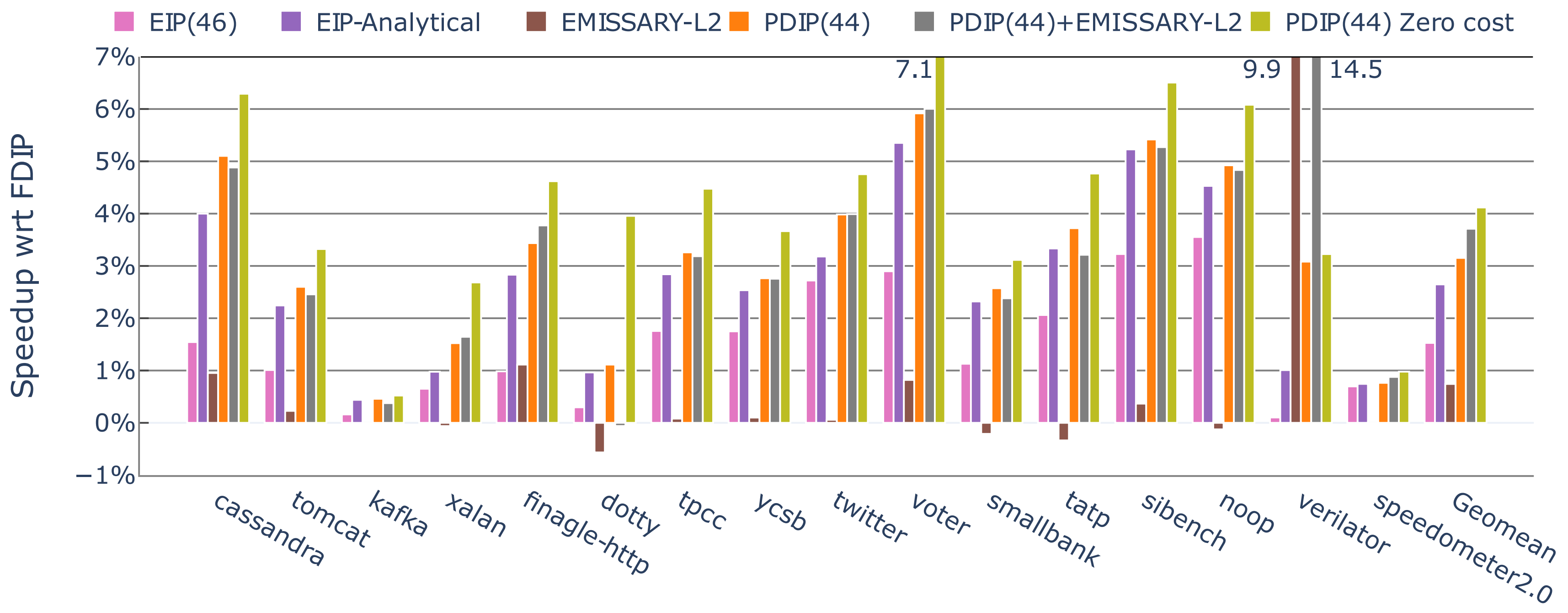


Figure 4.10: Speedup Comparison

resulting in a geomean speedup of 3.7%. The combination of PDIP(44)+EMISSARY-L2 thus captures 72.5% of FEC-Ideal, described in Section 4.3.

Since EMISSARY-L2 preserves instructions in L2, one drawback is that it causes contention for L2 data accesses. For example, the dotty, tatp, and smallbank benchmarks all show a considerable increase in L2 data MPKI with EMISSARY-L2 enabled. Thus, FEC lines stored in L2 can reduce the amount of space available for L2 data, resulting in performance degradation. Therefore, benchmarks with higher L2 data pressure could cause an increase in L2 data MPKI with EMISSARY-L2 enabled. This results in PDIP+EMISSARY-L2 having slightly lower performance than PDIP only in such scenarios. On the other hand, benchmarks like verilator with very low L2 data pressure are more complementary.

We also evaluated PDIP on SPEC17 [11] workloads. PDIP(44) shows a geomean speedup of 0.17%. It shows that PDIP, when used with traditional workloads that are not front-end heavy do not show any slowdown.

4.7.2 Prefetch Timeliness And Accuracy

To understand the timeliness of PDIP prefetches, we implement and compare with a zero-cost prefetch policy, where each prefetch request is served with zero cycle penalty and placed in L1I. A no-cost prefetch of PDIP(44) shows a geomean 4.11% gain over baseline. PDIP(44) achieves 76.58% of a zero-cost policy at the same hardware budget, which places a ceiling on lost performance due to partial misses. PDIP achieves at least 75% of zero cost policy even at larger table sizes. Further, Figure 4.11 shows the number of late prefetches (partial hits) issued by PDIP. On average, 12.6% of prefetches requests issued by PDIP are late, indicating that the heavy majority of prefetches are timely and contribute to the performance gain.

Table 4.4 captures the Mean Prefetch per kilo instructions (PPKI) of all policies and demonstrates the accuracy of prefetches of different PDIP configurations in comparison to EIP(46) and EIP-Analytical. Accuracy is defined as the % of prefetches that were accessed by a demand fetch before eviction. Thus, the prefetches must be useful and timely for high

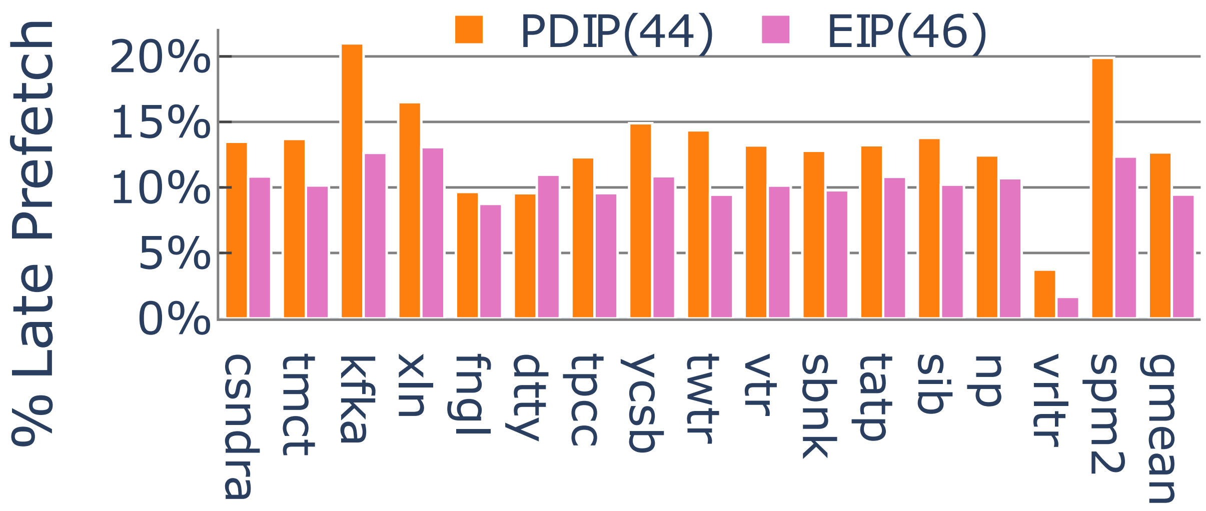


Figure 4.11: % of Late Prefetches per Benchmark in PDIP(44)

Metric	**EIP (46)**	**EIP Analytical**	**PDIP (11)**	**PDIP (44)**
PPKI	22	40	21	32
Accuracy	44%	45%	55%	54%

Table 4.4: Average Prefetch per Kilo Instructions (PPKI) and Prefetch Accuracy of all prefetch policies

accuracy. With large code footprints, too many inaccurate prefetches may evict useful lines in the L1I, leading to contention. Any discussion of prefetch effectiveness must also account for the rate of prefetches issued by the policy.

Across our benchmarks, PDIP prefetches, on average, show an accuracy of 55%, while the EIP policies show an accuracy of 45%. The preferred PDIP policy PDIP(44) has a PPKI of 32 and thus issues 45% more prefetches, yet is more accurate than EIP(46) at a similar storage budget. A storage-limited version, PDIP(11), issues the same number of prefetches as EIP(46) while being four times smaller and still maintaining higher accuracy. Thus, the PDIP configurations store more relevant metadata more efficiently. EIP-Analytical and EIP(46) have similar prefetch accuracy, but EIP-Analytical issues nearly two times more prefetch requests, putting more pressure on the L1I with more inaccurate prefetches.

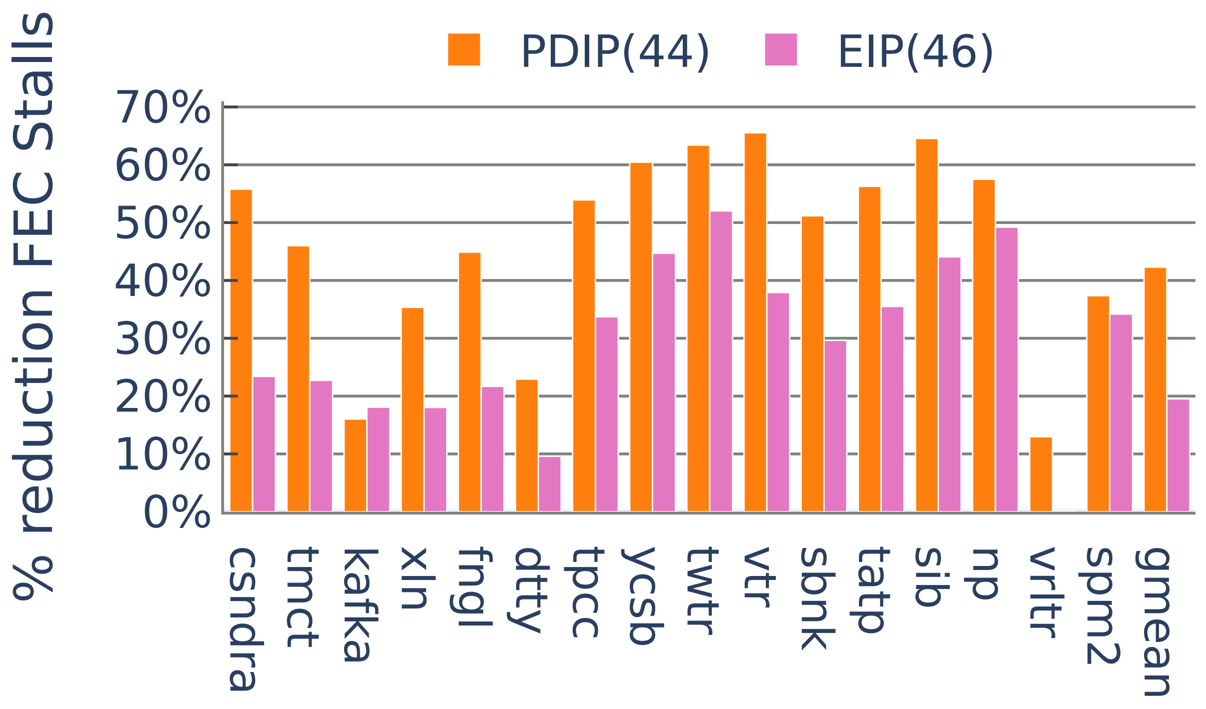


Figure 4.12: % reduction FEC stalls per benchmark in PDIP(44) and EIP(46)

4.7.3 Prefetch Effectiveness

As discussed in Section 4.3, L1I misses have a high variance in performance criticality, depending on whether or not they are exposed to the FDIP front-end. It was also found that a small number of lines contribute to the majority of front-end stalls. Thus, coverage of critical stalls is a better metric of comparative prefetch effectiveness than coverage of prefetch lines. For example, by prioritizing the criticality of lines, PDIP reduces FEC stalls by an average of 42%, compared to 19% with EIP for a similar hardware budget. In addition, Figure 4.12 shows that PDIP reduces FEC stalls by 50% or more in over nine benchmarks with high FEC line coverage. This translates to a reduction in total stalls of 16% for PDIP and 8% for EIP. Benchmarks with lower L1I pressure and fewer FEC lines (such as `kafka` and `speedometer2.0` as shown in Figures 4.9 & 4.4) show similar reductions in FEC stalls with PDIP and EIP. However, because PDIP maintains higher performance by focusing on fewer lines, it generates fewer prefetches and consequently half as many useless prefetches

(prefetches evicted without hits) as EIP. Thus, PDIP self-adjusts better in such benchmarks and causes less cache pollution than other methods. In contrast, in benchmarks with very high FEC pressure (such as `verilator`), PDIP aggressively targets FEC lines, generating over ten times as many prefetches as EIP, thus reducing FEC stalls by 12% compared to EIP's 0.05%. For such benchmarks, complementary techniques like EMISSARY-L2 work in tandem, reducing FEC stalls by 46% in the PDIP+EMISSARY-L2 configuration. Despite the focus on criticality, Section 4.7.2 shows that PDIP prefetches are still sufficiently timely, generating more accurate prefetches while covering more of the critical stalls as compared to other prefetchers and targeting fewer lines. For criticality-based prefetchers, instead of measuring total prefetch line coverage, we prefer to define coverage over front-end critical misses (the ones that actually impact performance) rather than all misses. Thus, our definition of coverage is the percentage of all FEC misses that are targeted by PDIP. On average, PDIP has over 67% coverage of FEC lines.

Prefetches issued by PDIP can evict cache lines that could potentially be used later. An aggressive prefetching scheme could evict all useful lines. A prefetcher needs to balance so that useful lines are not evicted. To measure the impact of PDIP prefetches on useful lines evicted, we measure them using a victim cache, which is the same size as the L1I. The victim cache is populated when the PDIP prefetch fill evicts a cache line that is not a PDIP prefetched cache line. The victim cache is used to measure the hits, not to serve any data. The ratio of hits in the victim cache to the total number of prefetches shows the impact of PDIP prefetches on useful cache lines. On average 30% of cache lines evicted by PDIP(44) are needed again later. It shows that the evicted cache lines are of lower priority; thus, they do not hurt performance when PDIP is enabled.

4.7.4 PDIP Table Sensitivity Analysis

We study the impact of scaling PDIP Table size on performance by varying the number of ways. We model PDIP tables from 11KB to 87KB by having fixed 512 sets and varying the associativity from 2 to 16. Figure 4.13 shows performance gain with respect to the FDIP baseline. We store up to two targets and four consecutive offsets per entry for all PDIP policies, as empirical analysis showed 95% of targets are stored with two targets per entry. PDIP shows strong scaling for the majority of benchmarks up to 43.5KB but then shows diminishing returns thereafter.

All benchmarks except `verilator` show either improved or the same performance with an increase in PDIP Table size. We used optimized (using Facebook's BOLT [61]) binary, which has unusually long basic blocks which don't fit in the PDIP Table well. In the case of `verilator`, increasing mask bits per entry shows better scaling than increasing the total number of entries.

4.7.5 Energy and Area Analysis

We modified McPAT [55] to model the PDIP structures to generate energy and area overheads. Table 4.5 shows the % increases in energy consumption and area overhead of the CPU core. As we can see, all the configurations provide sufficient speedups in relation to their energy and area overheads. PDIP(44) provides the right balance of resource usage and performance.

Metric	**PDIP(11)**	**PDIP(22)**	**PDIP(44)**	**PDIP(87)**
Energy	0.25%	0.55%	0.62%	0.64%
Area	0.31%	0.52%	0.96%	2.84%

Table 4.5: Percentage increase in CPU core Energy consumption and Area over baseline modeled in McPAT

4.7.6 BTB Sensitivity Analysis

The performance of the FDIP front-end depends critically on the accuracy of the BPU. Thus, PDIP should also be compared with alternative approaches to improve the front-end with more BPU resources. For large code footprints, BTB budget is the main bottleneck of

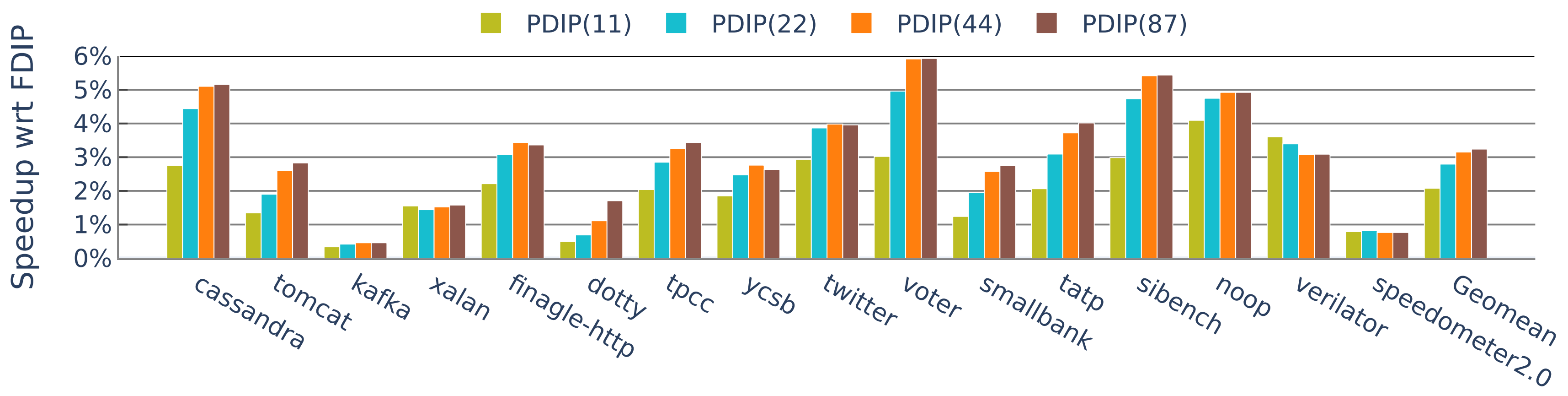


Figure 4.13: PDIP Policies with various PDIP Table configurations

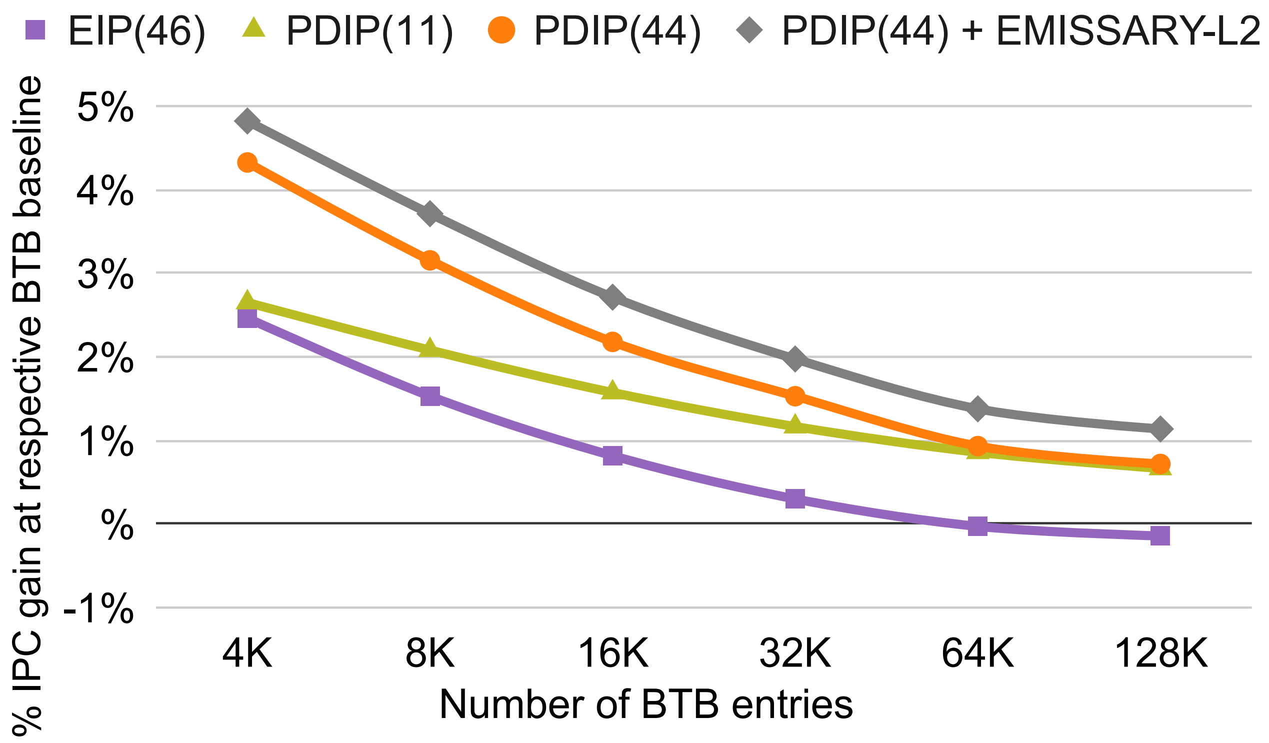


Figure 4.14: % IPC speedup of prefetch policies at various BTB sizes.

performance over BPU table sizes. Our experiments confirm this trend that the size of the BTB is a more important factor in scaling than the size of the BPU tables. When we model a large, highly accurate BPU that matches industry standards, we observe that scaling to larger BPU table sizes gives very little variation in results. In this section, we examine the effect of larger BTBs, both to (1) show the efficacy of PDIP even in the presence of future, aggressive BTBs and (2) to demonstrate that PDIP provides speedup in a much more area-efficient manner than BTB resizing alone. We are examining BTB sizes of < 8k entries, representative of efficiency cores, 8K-32K entries, representative of current and upcoming high-performance cores, and >32k entries, which correspond to future cores, and an extended examination for comprehensive insights. Figure 4.14 shows that at smaller BTB sizes, FDIP's poorer performance allows additional headroom for a prefetcher, and PDIP(44) captures most of this, showing 4.32% speedup at 4K-entry BTB (59KB) and 3.15% speedup at 8K-entry BTB (119KB) over FDIP at their respective BTB sizes. For larger BTB sizes, there is limited headroom available, so PDIP(11) and PDIP(44) converge. Also, since PDIP

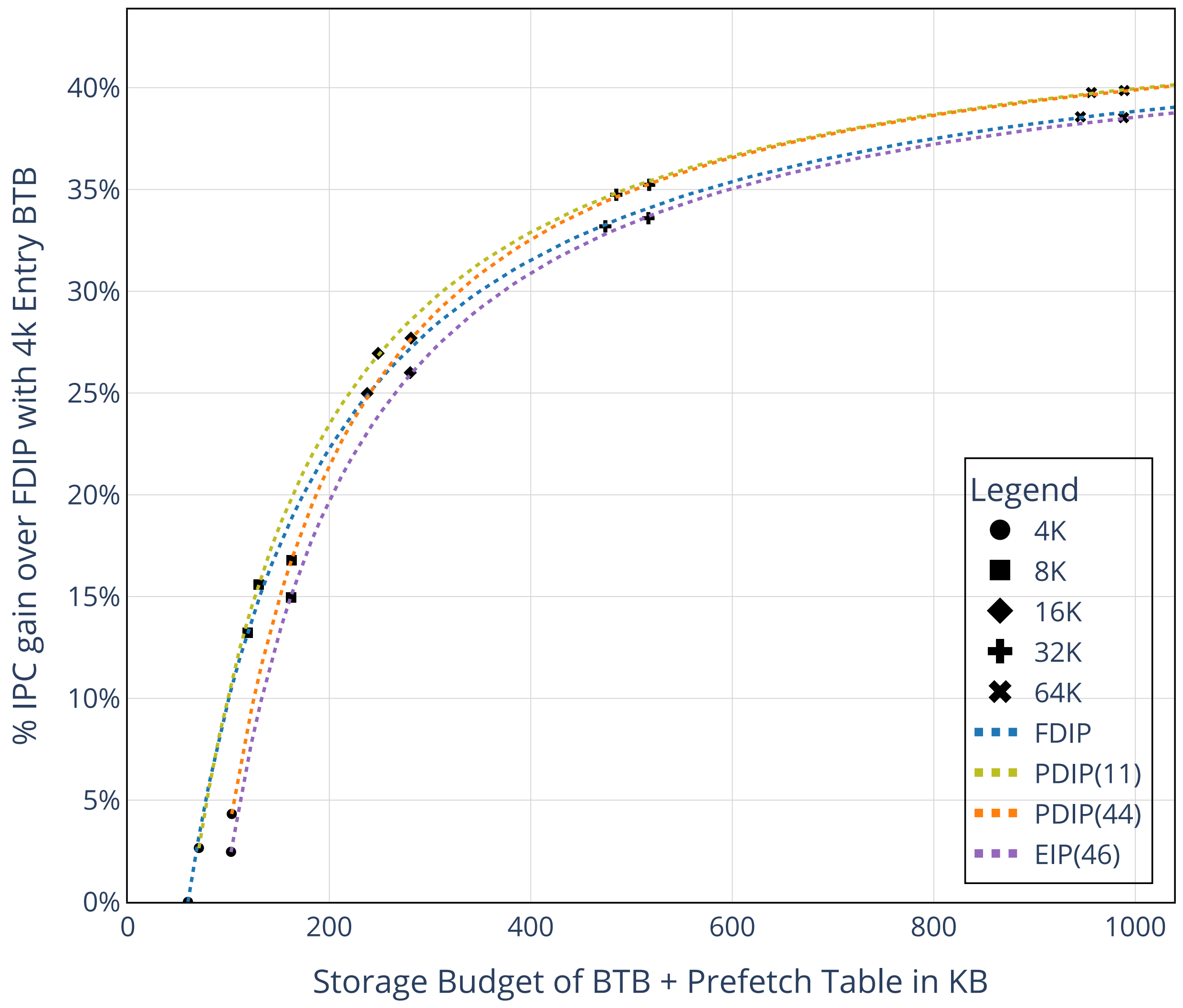


Figure 4.15: IPC performance gain across different policies at BTB sizes 4K (59KB), 8K (119KB), 16K (237KB), 32K (473KB), and 64K (945KB). PDIP(11), PDIP(44) and EIP(46) needs 10.875KB, 33.5KB and 46KB additional storage respectively.

uses the same tracking hardware as EMISSARY-L2 , PDIP paired with EMISSARY-L2 provides the most storage-efficient solution at any BTB size, but for brevity, it is omitted from the following discussion.

Figure 4.15 compares the storage effectiveness of a prefetcher as compared to scaling the BTB. It shows that one of the PDIP configurations always makes better use of storage than scaling the BTB at every stage. Conversely, EIP is always a more inefficient use of storage than increasing BTB size. At low BTB sizes, corresponding to efficiency cores, the smaller PDIP(11) provides higher-scaled performance. For PDIP(11) with 8k-entry BTB, FDIP would need 16KB additional BTB scaling as compared to PDIP's 11KB to match its performance. At larger BTB sizes, corresponding to high-performance cores, the higher performance of PDIP(44) is apparent. For PDIP(44) with 32k-entry BTB, FDIP would need 111KB additional BTB scaling as compared to 44KB of additional storage to match the same performance as PDIP. Thus, PDIP(44) uses 60% lesser additional storage. We see that PDIP (except in the case of a very small BTB) provides significantly more efficient use of storage than scaling the BTB, and these gains would improve when paired with EMISSARY-L2 . Furthermore, this also corroborates Ishii et al.' s[40] observation that prior prefetching techniques provide little performance improvement over modern FDIP machines with large BTBs [71, 17] as evidenced in Figure 4.14 with EIP. The criticality-aware nature of PDIP targets scenarios where FDIP fails and thus shows performance over FDIP regardless of the BTB size, showing more than 1.0% speedup even with a 64K-entry BTB (945KB).

4.7.7 Prefetch Triggers Analysis

A prefetch trigger in PDIP is always associated with a front-end stall-causing event, such as a branch mispredict or a full FTQ. In the former case, we use the mispredicted branch as the prefetch trigger, while in the latter case, we use the last taken branch instead. Here, we examine the distribution of the types of prefetch triggers that lead to a target being prefetched. Figure 4.16 shows that, on average, branch mispredictions contribute to 89% of the issued prefetch targets, while last taken branches contribute to only 11%.

4.8 Related Work

4.8.1 Hardware Instruction Prefetchers

EIP [69] proposed entangling, i.e., associating, of a cache miss causing line of a variable latency L, with an entry that was accessed L cycles prior. This association should allow it

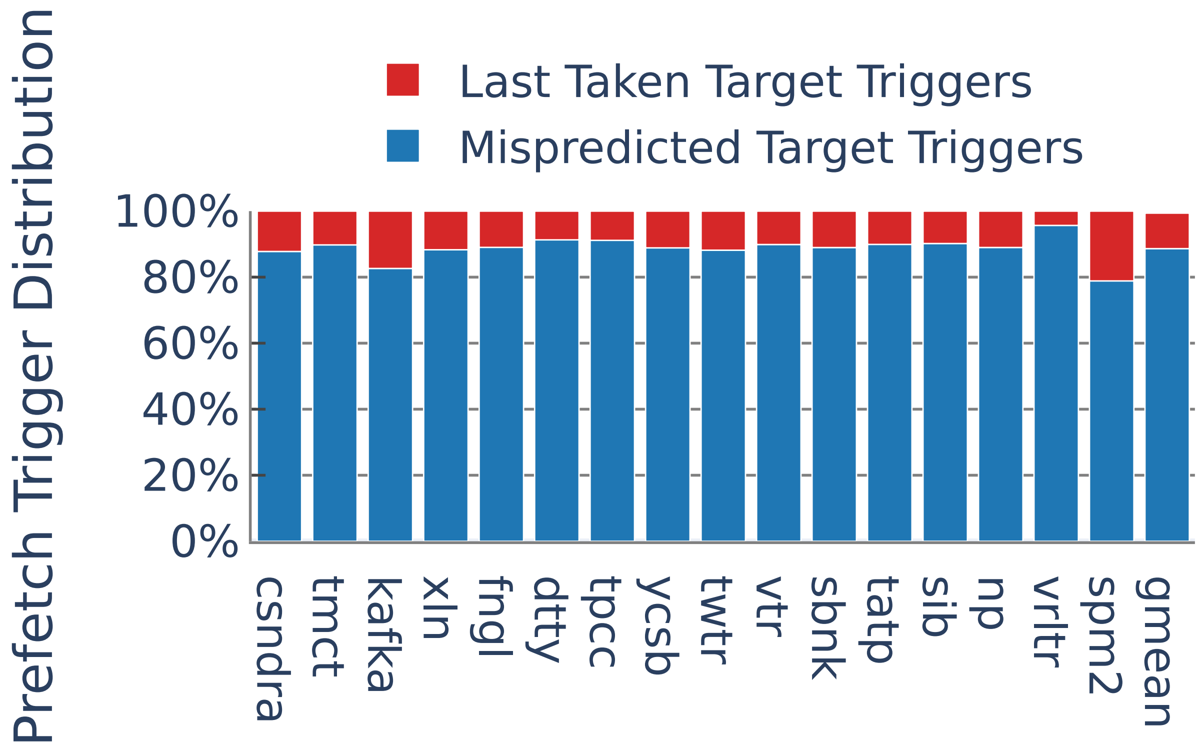


Figure 4.16: Distribution of prefetches based on Prefetch Trigger scenario

to prefetch it in a timely manner the next time the same line is accessed along the same execution path. Latency-based entangling could improve cache miss rate by prefetching lines long before they are used but may end up evicting cache lines, which are critical for improving performance. Other results [40] agree with ours that EIP does not show heavy improvement over an aggressive FDIP front-end.

FNL+MMA [75] prefetcher combines two techniques – Footprint Next Line(FNL) and Multiple Miss Ahead(MMA) prefetcher. FNL predicts the ”worth” of the next five consecutive blocks of a block B, which missed in a shadow I-Cache. A shadow I-Cache contains only tags and acts as a proxy for I-Cache misses. MMA predicts the block that is going to miss after a fixed number of n misses from the current block. The observation was that the same block is going to be missed again, and similarly, the next block will be used in the near future. This observation is similar to that of PDIP in that the same prefetch target and its associated branch trigger are going to cause bubbles in the pipeline again.

Several ”Record and Replay” techniques [33, 22, 47, 73] were proposed to prefetch instruction blocks well ahead of time using a history buffer that records the sequence of cache blocks. MANA [22] and PIF [33] (Proactive Instruction Fetch) were similar to the data cache prefetching technique [60]. These techniques take advantage of the temporal and spatial locality of the blocks accessed and store them in a table, which is accessed when a new instruction block is accessed; it then prefetches targets from the table. MANA proposes techniques to store target addresses in a storage-efficient way using High-Order-Bits-Patterns' Table(HOBPT). blocks which were significant in improving performance. PDIP Table can be augmented with HOBPT to address out-of-page entries efficiently.

Similarly SN4L+Dis+BTB [23] combines three techniques. SN4L handles contiguous blocks, Dis handles non-contiguous blocks, and BTB is an improvised Confluence[48] solution. Contiguous and non-contiguous blocks can be handled by FDIP as long as branch instructions found don't miss in BTB. Using a large enough BTB ensures that reused entries don't miss frequently in BTB, which leaves cold branch instructions. Our observation was that the majority of BTB misses are due to cold branch instructions. Jukebox [73] is specifically designed for serverless functions, which are short but incur high cache miss rates due to interleaved invocations. It records and replays to prefetch instructions to the L2 cache.

Temporal instruction fetch streaming (TIFS) [34] also works based on record and replay techniques. TIFS records the streams of blocks that were missed in L1-I and replays it when the first block in the pattern is seen again later. One of the key observations of TIFS is that streams of blocks cause misses to repeat. The Temporal Ancestry Prefetcher (TAP) [35] is another prefetcher that takes advantage of temporal locality. Unlike TIFS, TAP looks at all accesses instead of misses. A history of the last 14 PCs is maintained in the history buffer, and when a miss is observed, all the entries corresponding to the history buffer in the ancestry table are updated. Every time a new block is accessed, it is looked up in the ancestry table, and all its corresponding entries are prefetched. The hardware cost of implementing the temporal technique is reduced by tracking only temporal lines that caused misses rather than all lines. The overall cost of implementing TAP is still significant compared to the size of the instruction cache.

Context signature based prefetching techniques [59, 52] prefetch lines that were missed in the same context last time. RDIP [52] uses the return address stack(RAS) as the signature. The key observation is that the misses seen in a given context repeat next time, and the return address stack or calling context is used to capture the context. D-JOLT [59] is an improved technique that not only uses RAS as context but also captures the blocks that are accessed after long range and short range in a given context so as to send timely prefetch requests. Another key difference between RDIP and D-JOLT is the way the signature is generated. RDIP uses the whole RAS to compute the hash, whereas D-JOLT uses a FIFO of return addresses, which includes additional function calls and a number of returns that happened in reaching a given point in the execution.

Branch predictor based prefetching techniques [53, 38] prefetch instruction blocks following a branch using the predicted target. JIP [38] maintains a hierarchy of tables for a direct branch with fewer targets and an indirect branch with many targets. These targets are used to prefetch when a branch PC in the speculative path matches one of the tables. A confidence value is associated with targets to select only one path when more than one path is possible. Effectively, JIP mimics run ahead fetching without making changes to the branch predictor state.

SHIFT [47] is a storage-efficient implementation of history buffers that exploits spatial locality. It is specifically designed for applications running multiple threads that execute similar code. The storage space required for a large history buffer is optimized by virtualizing it (i.e., saving it in LLC). Since the LLC is shared by all cores, they take advantage of the history buffer stored in it and issue prefetch requests separately per core. Only one core updates the history buffer in LLC.

Prefetching along the wrong path is proposed in [63]. This can be effective for many workloads, but prefetching the wrong path for every conditional branch would be less effective (lead to unwanted cache pollution) in large code footprint workloads that put higher pressure on the L1I.

4.8.2 Software Instruction Prefetchers

Software prefetching techniques [25, 50, 56] typically require changes to the Instruction Set Architecture (ISA) such that instruction lines are prefetched well ahead of their use. These techniques involve inserting prefetch instructions in the code. Cooperative Prefetching [56] technique uses the compiler to automatically identify injection sites using static analysis. AsmDB [25] uses execution profile information to insert prefetches to improve accuracy. I-SPY [50] also uses profile information but encodes context information using a special instruction that issues prefetches only if the context matches, thereby reducing unnecessary prefetches when not required. It also proposes using coalesced prefetches wherever possible to reduce code bloat. Software prefetching techniques can be used to address cold misses, which hardware techniques fail to address. These techniques could be used along with PDIP to improve overall performance.

Chapter 5

Simulation Infrastructure

Datacenter applications, characterized by their server-client model, entail servers awaiting client requests, processing them, and responding accordingly. These server applications are designed for continuous operation, utilizing multiple threads to maximize available compute resources. With deep and intricate software stacks, these inherently multi-threaded applications require significant time to reach a steady state. However, simulation datacenter workloads using execution-driven simulators prove excessively time-consuming. To address this challenge, we've developed a robust framework named QPoints. QPoints efficiently accelerate simulation time by capturing the real system's application state and transferring its state to gem5 at a steady state. This chapter addresses the challenges associated with simulating multi-threaded workloads and highlights how QPoints effectively mitigates these obstacles.

5.1 Practical Simulation of Multi-threaded Workloads

Execution-driven simulators meticulously simulate the execution of each instruction and update every component affected by these instructions. Consequently, they tend to be slow, with simulating 100 million instructions in detailed mode in gem5 taking anywhere from 20 to 60 minutes, a stark contrast to the mere milliseconds it would take on a real machine.

Modern server workloads, often written in high-level languages like Java, are inherently multi-threaded. However, simulating such multi-threaded workloads in an execution-driven model poses three challenges. First, multi-threaded workloads require significantly more time to simulate meaningful work due to huge startup code. Second, simulator support for running multi-threaded workloads is often limited. Third, multi-threaded workloads frequently rely on system calls for inter-thread communication, necessitating the simulation of the system kernel, further increasing simulation time. To address these challenges, we have developed a robust framework called QPoints. QPoints takes the state of an application running on a real machine and generates gem5-compatible checkpoints. These checkpoints enable the application to be resumed from the same point in its execution, effectively sidestepping the need for time-consuming simulation and allowing for accurate analysis of multi-threaded workloads within the gem5 simulator.

5.1.1 Need for Simulating Kernel (Full System)

Simulating any real-world application would require support to execute system calls. The gem5 simulator has two modes of operation to support system calls. A mode in which the system calls are emulated is called System Emulation (SE) mode, and a mode in which the whole kernel is booted is called Full System (FS) mode.

The SE mode in gem5 works by intercepting system calls and simulating the functionality of a system call by making necessary changes to the state of the system. For example, consider a `brk` syscall. It is used to allocate memory on a heap. When this system call is executed in SE mode, gem5 sets the end of the heap to the location pointed by the `brk` syscall. The advantage of this mode is that it is very fast as it trades expensive system call mechanisms for equivalent emulation. One of the disadvantages of this approach is that every system call needs to be implemented in gem5. Which is very tedious and challenging when some system calls need to change the state of the kernel. Another disadvantage is that multi-threaded applications that use system calls to monitor a certain event in the system to wake them up become challenging to implement. The number of system calls supported by the gem5 community is good enough to simulate SPEC workloads but not multi-threaded workloads, which need thread synchronization and communication support.

The FS mode in gem5 supports booting a Linux kernel and loads services from a disk image. In this mode, all system calls are supported. Thus, it can simulate any benchmark. One disadvantage is that booting a kernel involves executing several trillions of instructions before actually getting to simulate the workload. This problem becomes worse when the workload takes several minutes on a native machine to reach the region of interest. Simulations could take several days to months in a detailed simulation model. The following section discusses ways to reduce the time spent reaching the region of interest.

5.1.2 Fast Mode Not Fast Enough

Simulating a workload that has several billions of instructions in a detailed mode could take several days to months. In order to reduce time spent in reaching a region of interest, the gem5 simulator supports a fast-mode CPU, which simulates instructions atomically at the rate of one instruction per cycle. This CPU is also known as the Atomic Simple CPU. Once the region of interest is reached, the simulation can be switched to detail mode CPU. Atomic Simple CPU is ten times faster than the detailed Out-of-Order (OOO) CPU but is still not fast enough. Table 5.1 shows simulation times of applications with various instruction counts. Booting a Linux kernel takes about 24 minutes on gem5 with Atomic CPU. An application that takes about 1 second on a native machine would take about 20 minutes on the simulator. In several workloads, the region of interest is after the first few minutes from the start of the application. Simulating workloads would take more than a day using this method.

The Atomic CPU is fast but not practical for workloads that take several days time to reach the region of interest. The gem5 simulator has support to execute code on the host machine directly by using Kernel Based Virtual Machine (KVM) feature of the Linux kernel. The gem5 CPU model with KVM mode operation is called KVMCPU. The support for KVM

CPU is very limited. KVM CPU model can be used only when the simulated system (guest) and host machine are using the same Instruction Set Architecture (ISA). Simulating ARM workloads requires building and running gem5 on an ARM host machine.

Application/ Host Runtime	**Instruction Count (Billions)**	**Approximate Simulation Time**
Linux Boot	2.4 B	∼24 min
1 second	2 B	∼20 min
1 minute	120 B	∼20 hours
10 minutes	1200 B	∼8 days

Table 5.1: Simulation times with Atomic CPU on gem5 compared to native runtime

5.1.3 QPoints

Simulators have adopted robust checkpoint-based solutions in order to avoid re-running the whole program just to study a small region of interest. Checkpoints contain the architectural state so that an application can be resumed from the same point. Therefore, studying new microarchitecture features becomes practical, provided the checkpoints for the applications are available. The gem5 simulator supports the checkpoint mechanism. However, checkpoints need to be collected using the gem5 simulator. This limitation makes it challenging to collect new checkpoints.

The cost of collecting new checkpoints is prohibitively high when the simulator is involved. This cost can be eliminated by collecting the state of an application on a real machine. The Lapidary [77] tool creates a gem5-compatible checkpoint by collecting the state of the application from a real machine. Lapidary collects the application's register and memory state by running the application on `gdb` (debugger). One of the key limitations is that the checkpoints collected by Lapidary only work with the SE mode of gem5. The kernel state cannot be captured, so it is not present in the memory state of the application. Some data center applications use event polling mechanisms to wake up sleeping threads. This is not possible in SE mode. Another limitation of this tool is that it was built for X86 workloads.

Thus, it cannot be used with ARM workloads. We have created an ARM port of Lapidary. However, due to the above limitations, it could not be used with datacenter workloads.

We have built a framework called QPoints, which creates a gem5-compatible Full System checkpoint using the QEMU [26] emulation tool. QPoints is the checkpointing framework that takes the state of a real machine and converts it into a gem5-compatible checkpoint. The gem5 simulator can resume the execution of the application from the same point where the checkpoint is created. QEMU is a cross-platform emulation tool. When the host platform has the same ISA as the emulated application, QEMU takes advantage of hardware acceleration using KVM or equivalent kernel features. QEMU does fast binary translation when running cross-platform applications. QEMU software emulation is much faster than gem5's Atomic CPU model. QEMU supports various platforms (x86, ARM, RISC-V). It also supports booting an operating system from an unmodified disk image. Which is useful when working with unmodified code bases and operating systems. QPoints works with an unmodified version of QEMU.

Creating a full system is very challenging compared to a System Emulation checkpoint. In Full System mode, the emulated system uses input and output devices for interaction and the disk device for storage. One of the key challenges of creating a portable Full System checkpoint is keeping the platform configuration and device mapping the same. Once a kernel boots, the device mapping is stored in the kernel data structures. When the Full System state is ported to another system with a different platform, it is more likely to fail due to different device mappings. Physical memory mapping is one example where virtual to physical memory maps are stored in kernel memory. When the physical address mapping is different, a page table lookup fails or gives the wrong address. Due to these issues, Full System checkpoints created in one version of gem5 may not be compatible with another version of gem5. The platform configuration and device mapping of QEMU and gem5 are different, so a standard platform configuration is needed to create checkpoints. Thus, we created a new platform configuration in gem5 that is compatible with QEMU to avoid making changes to QEMU.

A Full System checkpoint constitutes CPU register state, memory dump, device state, and device controller state. The device state of a storage device is the disk image file, and the controller state is the virtual PCI state modeled by QEMU. A disk device is needed to load a RAM disk and various start-up services that data center workloads need. A serial console is used to interact with the system, i.e., to send commands and get output. A serial console state does not necessarily need to be preserved in the checkpoint. Therefore, it is omitted from the checkpoint. Thus, a minimum state requires a CPU register, physical memory dump, disk image, and controller state. The CPU register state is obtained by using `gdb`. The disk controller state and memory dump are obtained using QEMU's command mode. The disk state is obtained by making a mere copy of the disk image.

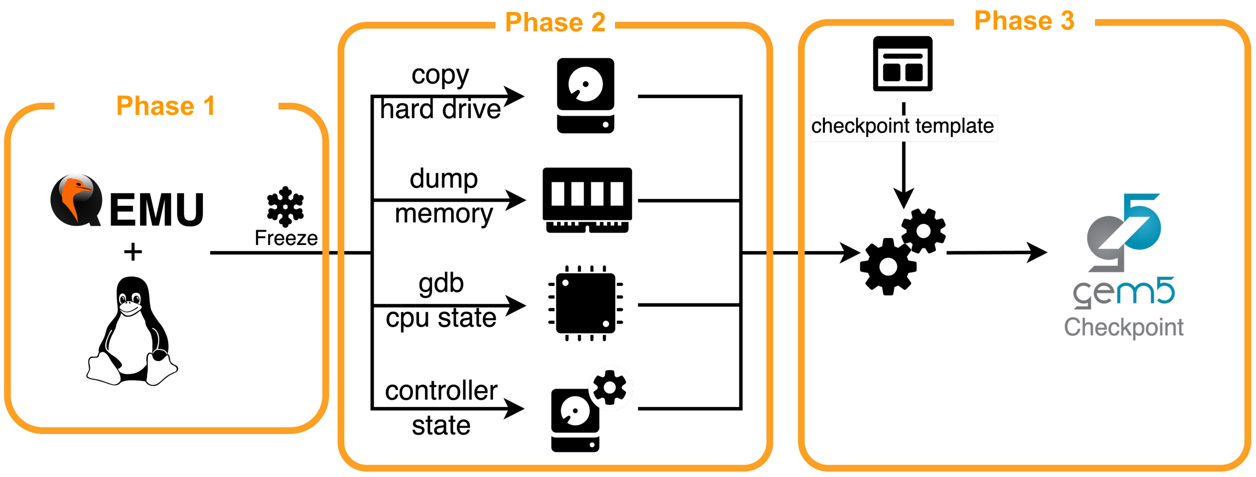


Figure 5.1: High Level Workflow of QPoints

Figure 5.1 shows the high-level workflow of the QPoints framework. The workflow of creating a checkpoint is divided into three phases. Phase one is In the application startup phase, the machine state is frozen, the state is extracted in phase two, and the gem5 checkpoint is generated in phase three.

Startup Phase (Phase One): This is the startup phase where the Linux System is booted using the QEMU's Full System emulation mode. When the host and guest use the same ISA, KVM acceleration can get better performance. Once the system is booted, execute a target benchmark using the console. Once the benchmark reaches the region of interest, the emulation is frozen, and all pending transactions are completed using QEMU's command-mode utility.

State Extraction (Phase Two): In this phase, the system state is extracted, which is then used to generate a gem5-compatible checkpoint. CPU register state is collected using the `gdb` debugging utility. QEMU provides a debugging feature where the emulated CPU state can be inspected using the `gdb` utility. The same technique is used to save the CPU register state. The physical memory dump is obtained using the QEMU's command mode utility. Similarly, the disk controller state is obtained using QEMU's command mode utility. Since the disk image is in the modified state, a copy of the disk image is created to preserve the disk state.

Checkpoint Generation (Phase Three): In this final phase, a gem5-compatible checkpoint is generated. The preceding phase yields a generic machine state, which gem5 cannot directly interpret. Gem5 offers a checkpoint feature for both Full System and System Emulation modes, typically stored as a single checkpoint file containing file paths to disk and memory images. The generic state is converted so that gem5's checkpointing utility understands it.

In enhancing the gem5 simulator to support the QPoints framework, several pivotal changes have been implemented. A fundamental aspect is the realization of device mapping within gem5 through the QEMU Virtual Configuration. This configuration is now exposed as a new system configuration, enabling more flexible and dynamic setups for simulation. Notably, GIC v2 has been enabled to specifically accommodate checkpoints collected using Apple's M1 hardware, ensuring compatibility with this architecture. Additionally, the disk is attached as a VirtIO device, providing an efficient and standardized interface for I/O operations. To maintain consistency and proper integration with gem5, it is crucial to adhere to a subset of VirtIO features during QEMU emulation. These modifications collectively contribute to a more versatile and robust gem5 simulator, aligning it with the requirements of the QPoints framework.

The QPoints framework boasts several advantages that significantly enhance its capabilities. It excels in executing intricate software with diverse requirements, effortlessly handling heavy runtime environments. The comprehensive support for all system calls ensures seamless compatibility with a wide range of applications. Notably, QPoints delivers near-native wait times, offering enhanced efficiency during simulation. One notable advantage is the elimination of the need to build gem5 on ARM systems. The framework's support for ARM-powered M1 Mac and free hardware acceleration beyond KVM further broaden its applicability. Importantly, these benefits are achieved without requiring any modifications to the QEMU source code. Additionally, the expanding horizons of hardware acceleration support, reaching beyond Linux to platforms like Apple's HVF (an equivalent of KVM), and the efficiency in handling gem5-compatible multi-core checkpoints are distinctive features that contribute to the framework's versatility and effectiveness.

QPoints, while offering valuable advantages, come with certain limitations that necessitate careful consideration. Firstly, its exclusive support for ARM 64-bit platforms may restrict its utility, particularly in environments with diverse architectures. The absence of symbol table mapping poses a challenge to debugging, making it difficult to trace and analyze program execution. Debugging complexities are further exacerbated by other factors, potentially impeding the efficient identification and resolution of issues within the framework. Moreover, QPoints lacks compatibility with gem5's COW and QEMU's QCOW2 image formats, limiting its integration with widely used storage formats. An additional constraint lies in each checkpoint containing a full disk image, leading to larger checkpoint files and increased storage requirements.

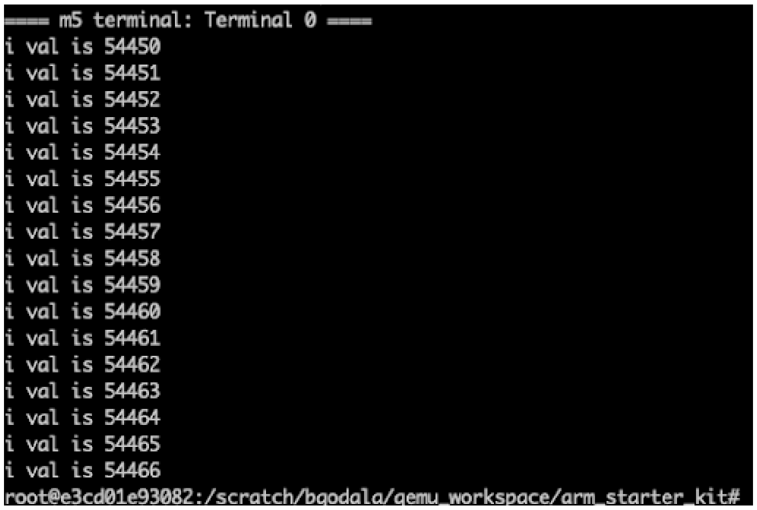
5.1.4 Correctness

The QPoints framework underwent rigorous testing to ensure its robustness and adherence to expected behaviors. Our comprehensive testing strategy encompassed a diverse range of unit tests meticulously designed to evaluate correctness and performance. Figure 5.2 shows the snapshot of the output of the first unit test when resumed on gem5. The output shows that gem5 can continue execution from the point where the checkpoint was collected. Multi-thread support was specifically scrutinized by executing a multi-threaded program, where the simulated program was anticipated to switch software threads seamlessly. Figure 5.4 shows the snapshot of the gem5 output of the multi-threaded test. This demonstrates that gem5 is able to simulate software context switches. This dynamic test workload included a disk integrity check, verifying the accurate listing of all files on the attached disk. Figure 5.3 shows the snapshot of the output generated by the gem5 simulation.

Extensive testing was conducted to assess correctness, allowing the program to run to completion in an Atomic CPU, followed by a meticulous comparison of the respective outputs. Furthermore, our evaluation extends to a rich collection of checkpoints sourced from renowned benchmark suites, including DaCapo, Renaissance, Cloudsuite v4, OLTP bench, and Speedometer 2.0. We are pleased to make this collection of checkpoints publicly available, providing a valuable resource for the research community [10]. This rigorous testing and the availability of diverse checkpoints underscore the reliability and applicability of the QPoints framework across various workloads and use cases.

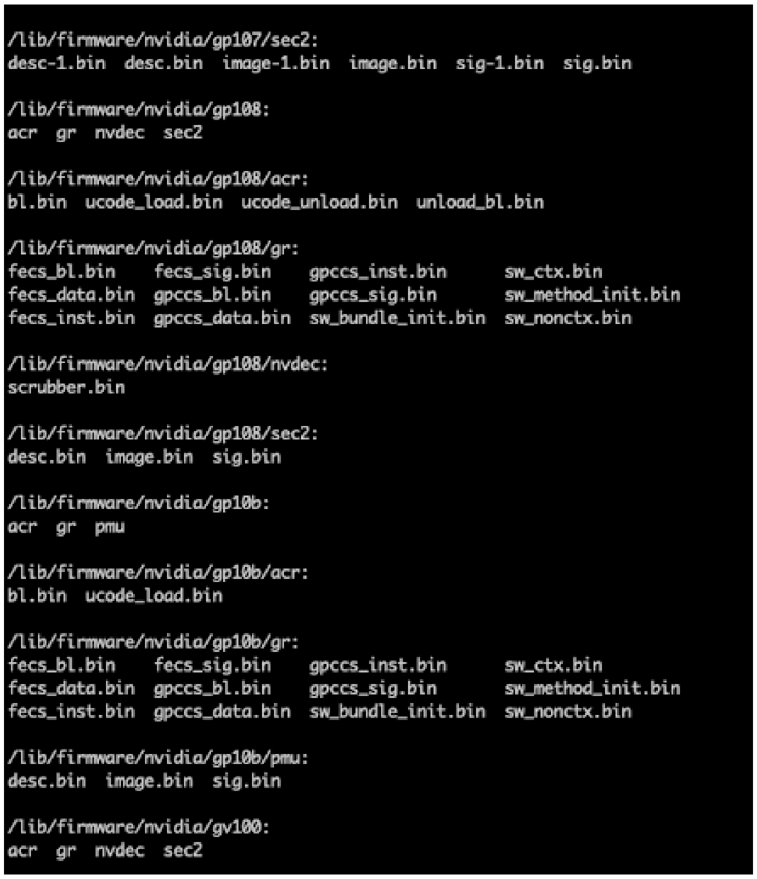
5.1.5 Techniques to Reduce Noise from OS and IO Events

Simulating workloads in the Full System mode presents a unique set of challenges, particularly concerning the context switching of software applications, which is governed by the simulated kernel. Additionally, the exposure of IO access bottlenecks introduces potential variability in simulation results, even with minor alterations to simulated hardware parameters. This complexity makes it challenging to attribute performance gains or losses specifi-



```
==== m5 terminal: Terminal 0 ====
i val is 54450
i val is 54451
i val is 54452
i val is 54453
i val is 54454
i val is 54455
i val is 54456
i val is 54457
i val is 54458
i val is 54459
i val is 54460
i val is 54461
i val is 54462
i val is 54463
i val is 54464
i val is 54465
i val is 54466
root@e3cd01e93082:/scratch/bgodala/qemu_workspace/arm_starter_kit#
```

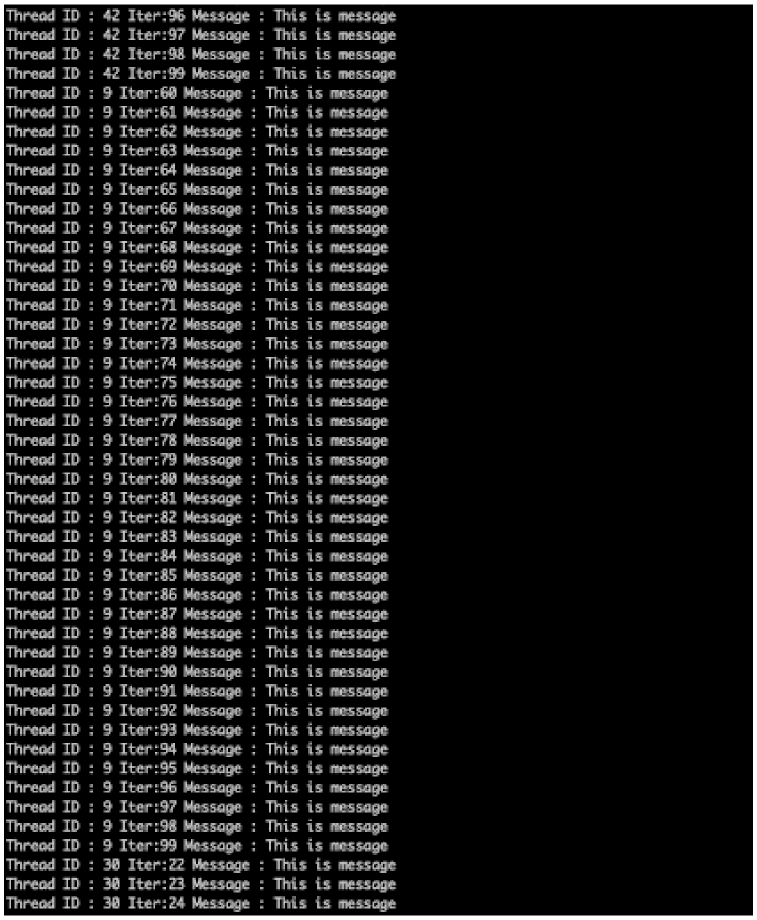
Figure 5.2: Output of a first test after resuming a QPoints checkpoint on gem5



```
/lib/firmware/nvidia/gp107/sec2:
desc-1.bin  desc.bin  image-1.bin  image.bin  sig-1.bin  sig.bin

/lib/firmware/nvidia/gp108:
acr  gr  nvdec  sec2

/lib/firmware/nvidia/gp108/acr:
bl.bin  ucode_load.bin  ucode_unload.bin  unload_bl.bin

/lib/firmware/nvidia/gp108/gr:
fecs_bl.bin    fecs_sig.bin    gpccs_inst.bin      sw_ctx.bin
fecs_data.bin  gpccs_bl.bin    gpccs_sig.bin       sw_method_init.bin
fecs_inst.bin  gpccs_data.bin  sw_bundle_init.bin  sw_nonctx.bin

/lib/firmware/nvidia/gp108/nvdec:
scrubber.bin

/lib/firmware/nvidia/gp108/sec2:
desc.bin  image.bin  sig.bin

/lib/firmware/nvidia/gp10b:
acr  gr  pmu

/lib/firmware/nvidia/gp10b/acr:
bl.bin  ucode_load.bin

/lib/firmware/nvidia/gp10b/gr:
fecs_bl.bin    fecs_sig.bin    gpccs_inst.bin      sw_ctx.bin
fecs_data.bin  gpccs_bl.bin    gpccs_sig.bin       sw_method_init.bin
fecs_inst.bin  gpccs_data.bin  sw_bundle_init.bin  sw_nonctx.bin

/lib/firmware/nvidia/gp10b/pmu:
desc.bin  image.bin  sig.bin

/lib/firmware/nvidia/gv100:
acr  gr  nvdec  sec2
```

Figure 5.3: Output of disk read test after resuming a QPoints checkpoint on gem5

cally to the evaluated hardware features. To mitigate these challenges, we employed various



```
Thread ID : 42 Iter:96 Message : This is message
Thread ID : 42 Iter:97 Message : This is message
Thread ID : 42 Iter:98 Message : This is message
Thread ID : 42 Iter:99 Message : This is message
Thread ID : 9 Iter:60 Message : This is message
Thread ID : 9 Iter:61 Message : This is message
Thread ID : 9 Iter:62 Message : This is message
Thread ID : 9 Iter:63 Message : This is message
Thread ID : 9 Iter:64 Message : This is message
Thread ID : 9 Iter:65 Message : This is message
Thread ID : 9 Iter:66 Message : This is message
Thread ID : 9 Iter:67 Message : This is message
Thread ID : 9 Iter:68 Message : This is message
Thread ID : 9 Iter:69 Message : This is message
Thread ID : 9 Iter:70 Message : This is message
Thread ID : 9 Iter:71 Message : This is message
Thread ID : 9 Iter:72 Message : This is message
Thread ID : 9 Iter:73 Message : This is message
Thread ID : 9 Iter:74 Message : This is message
Thread ID : 9 Iter:75 Message : This is message
Thread ID : 9 Iter:76 Message : This is message
Thread ID : 9 Iter:77 Message : This is message
Thread ID : 9 Iter:78 Message : This is message
Thread ID : 9 Iter:79 Message : This is message
Thread ID : 9 Iter:80 Message : This is message
Thread ID : 9 Iter:81 Message : This is message
Thread ID : 9 Iter:82 Message : This is message
Thread ID : 9 Iter:83 Message : This is message
Thread ID : 9 Iter:84 Message : This is message
Thread ID : 9 Iter:85 Message : This is message
Thread ID : 9 Iter:86 Message : This is message
Thread ID : 9 Iter:87 Message : This is message
Thread ID : 9 Iter:88 Message : This is message
Thread ID : 9 Iter:89 Message : This is message
Thread ID : 9 Iter:90 Message : This is message
Thread ID : 9 Iter:91 Message : This is message
Thread ID : 9 Iter:92 Message : This is message
Thread ID : 9 Iter:93 Message : This is message
Thread ID : 9 Iter:94 Message : This is message
Thread ID : 9 Iter:95 Message : This is message
Thread ID : 9 Iter:96 Message : This is message
Thread ID : 9 Iter:97 Message : This is message
Thread ID : 9 Iter:98 Message : This is message
Thread ID : 9 Iter:99 Message : This is message
Thread ID : 30 Iter:22 Message : This is message
Thread ID : 30 Iter:23 Message : This is message
Thread ID : 30 Iter:24 Message : This is message
```

Figure 5.4: Output of a multi-threaded test program after resuming a QPoints checkpoint on gem5

techniques to minimize noise and introduced new counters to ensure that any remaining noise falls within an acceptable error range.

One strategy involved leveraging the Linux kernel's `isolcpu` feature to isolate specific CPUs from consideration by the Linux scheduler. Consequently, no applications are scheduled on these isolated CPUs unless explicitly mapped using the `taskset` utility, which sets process affinity. This ensures that a process mapped onto an isolated CPU remains uninterrupted by the kernel, except for timer interruptions. Although periodic switching still occurs for multi-threaded applications mapped onto an isolated CPU, the Read Copy Update (RCU) the feature is not compatible with this method.

Another approach to minimize scheduler-induced noise is to assign the application the highest priority in the scheduling queues. Scheduling priority can be increased using the `nice` utility in the Linux system, with a scheduling priority value of -20 providing the highest priority. By setting the applications' scheduling priority to be on par with Linux kernel tasks that use the real-time scheduling policy, interference from lower-priority processes is reduced.

While gem5's process switching mechanism could introduce variability due to its dependence on the simulated timer circuit, we addressed this issue by implementing an instruction count-based switching mechanism. The process is switched either upon reaching a fixed instruction count or executing an exit system call. These refined techniques effectively kept the variability in simulation results within a narrow margin of 0.2%, measured as the change in the number of committed operations.

Chapter 6

Future Work

This chapter explores potential methods to extend criticality awareness to various front-end cache structures. The improved EMISSARY-L2 replacement policy, known for its criticality awareness, can be extended to structures like the Branch Target Buffer (BTB), Instruction Translation Lookaside Buffer (iTLB), Uop Cache, and others, as they all utilize replacement policies.

However, criticality-aware policies may not always lead to performance gains across all workloads. In such scenarios, dynamically adjusting criticality-aware policies based on workload characteristics could yield better results. By dynamically adapting the criticality-aware policies to the specific needs and behaviors of different workloads, the system can optimize cache utilization more effectively, ultimately leading to improved overall performance. This approach allows for flexibility in optimizing front-end cache structures to accommodate a wide range of workloads and their unique requirements.

6.1 Dynamic EMISSARY

The performance improvement achieved by EMISSARY-L2 can vary based on the maximum number of ways (N) used to preserve instruction lines and the randomness factor (r). On average, the best performance is observed when N is set to 8 and r is 1/32. However, individual benchmarks may exhibit peak performance at different values of N and r. Interestingly, benchmarks that are constrained by the front-end tend to show better performance when N exceeds eight ways (out of a total of sixteen ways), which suggests that dynamically adjusting the values of N and r based on the characteristics of workloads could lead to optimal performance gains. Furthermore, when enabling EMISSARY-L2 does not provide significant benefits, it can be turned off to mitigate its impact on overall performance. By dynamically adjusting these parameters and selectively enabling or disabling EMISSARY-L2 , the system can effectively adapt to varying workload demands and maximize performance efficiency.

6.2 EMISSARY for Uop Cache

In the x86 Instruction Set Architecture (ISA), instruction encodings are of variable lengths, with the actual lengths becoming known only after instruction decoding. Speculative parallel decode techniques are often employed to increase the decode width, but these methods consume significant power. To address this issue and simultaneously enhance width, modern processors from Intel and AMD utilize a Decoded Micro-Operation (Uop) cache.

While the Uop cache exhibits good utilization in SPEC workloads, its effectiveness diminishes in datacenter workloads due to the presence of large code footprints. The low utilization of Uop Cache is primarily because the cache replacement policy prioritize frequently used code over low-reuse code, resulting in low Uop cache utilization in datacenter workloads that have very large code footprints.

To mitigate this issue and optimize Uop cache usage, one potential approach is to repurpose the cache to preserve critical lines rather than prioritize high-frequency lines. By reallocating Uop cache resources to store and retain critical lines, the processor can better support the execution of datacenter workloads, where the importance lies more in preserving critical code segments rather than maximizing cache hits based on frequency alone. This strategy can potentially lead to improved performance and efficiency in datacenter environments.

6.3 Proritize iTLB miss events

PDIP is currently limited to branch resteer and FTQ full front-end events. However, similar to branch resteer events, a miss in the Instruction Translation Lookahead Buffer (iTLB) followed by a miss in the instruction cache is also considered critical.

When an iTLB miss occurs, the front-end stalls, even if the Fetch Queue (FTQ) contains valid entries. Thus, an iTLB miss has a serializing effect on instruction fetch, impacting overall performance. Prefetching cache lines that fall along the iTLB miss path could potentially alleviate this performance bottleneck. By proactively prefetching instructions that are likely to be needed due to the iTLB miss, the front-end can mitigate the stall time and improve overall performance efficiency.

Incorporating this additional prefetching mechanism into PDIP could further enhance its effectiveness in optimizing instruction fetch and improving performance in scenarios where iTLB misses occur frequently.

Chapter 7

Conclusion

In conclusion, this dissertation has presented two novel contributions, EMISSARY-L2 and PDIP, which collectively embody the essence of criticality-aware front-end designs for high-performance processors.

EMISSARY-L2 is an improved EMISSARY-L1 policy specifically designed for datacenter workloads. Observing that modern architectures completely tolerate many instruction cache misses, EMISSARY-L2 prioritizes, with persistence, inserted lines whose misses cause decode starvation over those whose misses did not. Without the need to track history, coordinate with prefetchers, make predictions, or perform complex calculations, EMISSARY-L2 consistently improves performance and saves energy while remaining simple to implement. EMISSARY-L2 shows a geomean performance gain of 3.24% (up to 23.7%) and a geomean energy savings of 2.12% (up to 17.7%) over TPLRU on top of a state-of-the-art FDIP prefetcher to model the aggressive front-ends found in modern processors. This speedup is 21.6% of the total speedup obtained by an unrealizable model with an ideal L2 instruction cache with a mere 4KB hardware budget.

A Criticality-Aware Instruction prefetcher, PDIP, is carefully designed to complement a decoupled front-end, which prefetches FEC lines where FDIP struggles. PDIP recognizes that most instruction cache lines are effectively fetched by FDIP in time to completely hide any front-end stalls. Thus, PDIP only targets the fraction of Instruction Cache misses that are not hidden by FDIP. Thus, it provides higher coverage of the performance-critical misses and higher accuracy than other prefetchers. It functions in synergy with EMISSARY-L2 and achieves up to 72.5% of FEC-Ideal.

Bibliography

[1] Apache cassandra. http://cassandra.apache.org/.

[2] Apache kafka. https://kafka.apache.org/.

[3] Apache Solr. https://solr.apache.org/.

[4] Apache tomcat. https://tomcat.apache.org/.

[5] Browserbench. ”https://browserbench.org”.

[6] Dotty scala compiler. ”https://github.com/lampepfl/dotty”.

[7] Intel VTune. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[8] MediaWiki. https://www.mediawiki.org/wiki/MediaWiki.

[9] Postgresql. ”https://www.postgresql.org/”.

[10] QPoints ARM Worklaods used in EMISSARY. https://drive.google.com/file/d/1ac6OR-nuENQjw-rRBR-0S9rYQEEuCvyp/view?usp=drive_link.

[11] Spec newsletter. ”https://www.spec.org”.

[12] Specjbb 2015. ”https://www.spec.org/jbb2015/”.

[13] Speedometer2.0. ”https://browserbench.org/Speedometer2.0/”.

[14] TPC-C. http://www.tpc.org/tpcc/.

[15] Twitter finagle. https://twitter.github.io/finagle/.

[16] Verilator. https://www.veripool.org/wiki/verilator.

[17] Wikichip. https://en.wikichip.org/wiki/intel/microarchitectures/golden_cove.

[18] Ycsb. ”https://github.com/brianfrankcooper/YCSB/”.

[19] Champsim Simulator. https://github.com/ChampSim/ChampSim, 2020.

[20] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R. Prasky, and Anthony 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, 2020.

[21] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[22] Ali Ansari, Fatemeh Golshan, Rahil Barati, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. Mana: Microarchitecting a temporal instruction prefetcher. *IEEE Transactions on Computers*, 72(3):732–743, 2023.

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

[24] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[25] Grant Ayers, Nayana Prasad Nagendra, David I. August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers. In *International Symposium on Computer Architecture (ISCA)*, 2019.

[26] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.

[27] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[28] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In

Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA ’06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[29] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, dec 2013.

[30] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *2012 45Th annual IEEE/ACM international symposium on microarchitecture*, pages 389–400. IEEE, 2012.

[31] Stijn Eyerman, Sam Van Den Steen, Wim Heirman, and Ibrahim Hur. Simulating wrong-path instructions in decoupled functional-first simulation. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 124–133, 2023.

[32] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Not.*, 47(4):37–48, mar 2012.

[33] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. Proactive instruction fetch. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 152–162, 2011.

[34] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Temporal instruction fetch streaming. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, page 1–10, USA, 2008. IEEE Computer Society.

[35] Nathan Gober, Gino Chacon, Daniel A. Jiménez, and Paul V. Gratz. The temporal ancestry prefetcher. 2020.

[36] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A. Pokam, Jared Stark, Andre Seznec, Dean Tullsen, and David I. August. Pdip: Priority directed instruction prefetching. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS ’24, page 846–861, New York, NY, USA, 2024. Association for Computing Machinery.

[37] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51, 2020.

[38] Vishal Gupta, Neelu Shivprakash Kalani, and Biswabandan Panda. Runjump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching. *The First Instruction Prefetching Championship*, 2020.

[39] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166), feb 2008.

[40] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. Rebasing instruction prefetching: An industry perspective. *IEEE Computer Architecture Letters*, 19(2):147–150, 2020.

[41] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam 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, 2021.

[42] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 343–353. IEEE, 2015.

[43] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *37th International Symposium on Computer Architecture (ISCA)*, 2010.

[44] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Computer Architecture (ISCA)*, 2015.

[45] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169, 2015.

[46] Harshad Kasture and Daniel Sanchez. TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In *Workload Characterization (IISWC)*, 2016.

[47] Cansu Kaynak, Boris Grot, and Babak Falsafi. Shift: Shared history instruction fetch for lean-core server processors. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 272–283, 2013.

[48] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: Unified instruction supply for scale-out servers. In *Microarchitecture (MICRO)*, 2015.

[49] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris 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.

[50] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. I-spy: Context-driven conditional instruction prefetching with coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–159, 2020.

[51] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. Ripple: Profile-guided instruction cache replacement for data center applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 734–747, 2021.

[52] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. RDIP: Return-address-stack directed instruction prefetching. In *Microarchitecture (MICRO)*, 2013.

[53] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[54] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. Boomerang: A metadata-free architecture for control flow delivery. In *High Performance Computer Architecture (HPCA)*, 2017.

[55] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–480, 2009.

[56] Chi-Keung Luk and Todd C Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Microarchitecture (MICRO)*, 1998.

[57] Nayana Prasad Nagendra. *IMPROVING INSTRUCTION CACHE PERFORMANCE FOR MODERN PROCESSORS WITH GROWING WORKLOADS*. PhD thesis, Princeton University, 2021.

[58] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A. Pokam, Simone Campanoni, and David I. August. EMISSARY: Enhanced Miss Awareness Replacement Policy for L2 Instruction Caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA*. ACM, 2023.

[59] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. D-jolt: Distant jolt prefetcher. *The 1st Instruction Prefetching Championship (IPC1)*, 2020.

[60] K.J. Nesbit and J.E. Smith. Data cache prefetching using a global history buffer. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 96–96, 2004.

[61] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[62] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro*, 40(2):53–62, 2020.

[63] Jim Pierce and Trevor Mudge. Wrong-path instruction prefetching. In *Microarchitecture (MICRO)*, 1996.

[64] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: Benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel Emer. Adaptive insertion policies for high performance caching. In *34th International Symposium on Computer Architecture (ISCA)*, 2007.

[66] 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, 1999.

[67] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *Microarchitecture (MICRO)*, 1999.

[68] Glenn Reinman, Brad Calder, and Todd Austin. Optimizations enabled by a decoupled front-end architecture. *Computers, IEEE Transactions on*, 50:338 – 355, 05 2001.

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

[70] Alberto Ros and Alexandra Jimborean. Wrong-path-aware entangling instruction prefetcher. *IEEE Transactions on Computers*, 73(2):548–559, 2024.

[71] J Rupley. Samsung exynos m3 processor. *IEEE Hot Chips*, 30, 2018.

[72] Jeff Rupley, Brad Burgess, Brian Grayson, and Gerald D Zuraski. Samsung m3 processor. *IEEE Micro*, 39(2):37–44, 2019.

[73] David Schall, Artemiy Margaritov, Dmitrii Ustiugov, Andreas Sandberg, and Boris Grot. Lukewarm serverless functions: Characterization and optimization. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 757–770, New York, NY, USA, 2022. Association for Computing Machinery.

[74] André Seznec. A 64-kbytes ittage indirect branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

[75] André Seznec. The fnl+mma instruction cache prefetcher. 2020.

[76] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *Journal of Instruction-level Parallelism - JILP*, 8, 02 2006.

[77] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 572–586, New York, NY, USA, 2019. Association for Computing Machinery.

[78] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[79] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44. IEEE, 2014.