PDIP: Priority Directed Instruction Prefetching

Bhargav Reddy Godala
Princeton University
USA
bgodala@princeton.edu

Sankara Prasad Ramesh
University of California, San Diego
USA
spramesh@ucsd.edu

Gilles A. Pokam
Intel Corporation
USA
gilles.a.pokam@intel.com

Jared Stark
Intel Corporation
USA
jared.w.stark@intel.com

Andre Seznec
Intel Corporation
France
andre.seznec@intel.com

Dean Tullsen
University of California, San Diego
USA
tullsen@ucsd.edu

David I. August
Princeton University
USA
august@princeton.edu

Abstract

Modern server workloads have large code footprints which 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 reducing 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 paper, we propose Priority Directed Instruction Prefetching (PDIP), a novel 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.

ACM Reference Format:
Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A. Pokam, Jared Stark, Andre Seznec, Dean Tullsen, and David I. August. 2024. PDIP: Priority Directed Instruction Prefetching. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620665.3640394

ACM ISBN 979-8-4007-0385-0/24/04
https://doi.org/10.1145/3620665.3640394

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 [19, 29]. Applications with such large code footprints are typically dominated by front-end bottlenecks, as shown in Figure 1, which analyzes one important, representative workload (top-down analysis [53] obtained on Alderlake desktop CPU using Intel's VTune profiler [6]). 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 (L1-I), with capacity misses inducing a large number of stalls [29] 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 L1-I sits on the critical path. Prefetching has the potential to address this bottleneck at a lower cost [19, 23, 24, 30, 32, 37, 49]. However, these techniques have been less effective for datacenter and cloud workloads which exhibit instruction footprints several orders of magnitude larger than traditional server applications [31, 34, 35]. Modern processors implement a decoupled front-end, aka fetch directed instruction prefetching (FDIP) [45], as an attempt to remedy this problem [15, 26, 42, 48]. With FDIP, the L1-I fill is disassociated from its demand access thus allowing the front-end to aggressively prefetch along the

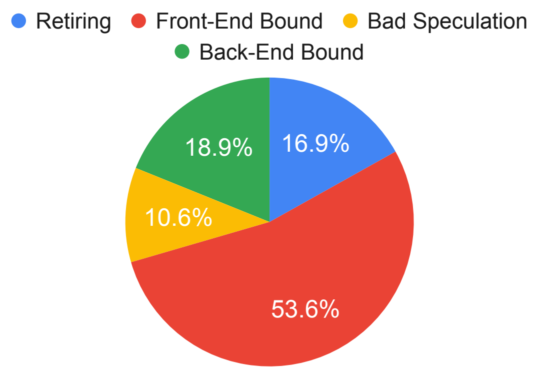

Figure 1. Top-down issue slots breakdown of cassandra benchmark on Alderlake host machine.

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 L1-I misses. FDIP dramatically changes both the access/miss pattern seen by the L1-I, and the criticality of misses (some misses are completely hidden by FDIP, others are not). Recent work on instruction prefetching [17, 18, 25, 27, 33, 39, 46, 51] have shown they can be effective in reducing misses, but either fail to account for the existence of FDIP or the variance in 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 paper 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 [38]) 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 paper 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) [46]. 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 [12], 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 [38], PDIP is still able to provide great value, realizing a geomean IPC of 3.7% across all benchmarks.

2 Background

This section provides background knowledge of decoupled front-end microarchitectures and their implications on instruction prefetching techniques. It also describes prior work [38] on front-end criticality aware cache replacement.

2.1 Decoupled Front-end

Figure 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 in to the L1-I, thus non-resident instruction blocks can be prefetched into the L1-I 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 back-end, keeping the FTQ full in the absence of squashes in the pipeline.

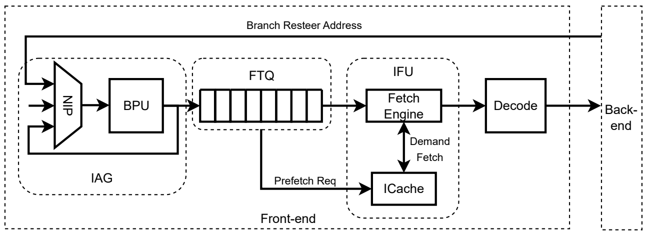

Figure 2. Generic decoupled front-end microarchitecture

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 L1-I 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 back-end, 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 of prefetching and tolerate cache miss latency. The importance of FTQ in the context of instruction prefetching was amply discussed in [28]. The authors show that most of the performance benefits obtained with recent instruction prefetching proposals [46] 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 L1-I misses. Some misses have no impact on front-end (or, naturally, back-end) 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.

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.

The EMISSARY [38] cache replacement policy identifies lines as front-end critical, and gives such lines priority in the replacement policy.

The main idea behind EMISSARY 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 work) 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 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 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 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 [38] 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.

3 Do We Need Another Prefetcher?

EMISSARY seeks to remove the most damaging (i.e., front-end critical) misses from the front-end, and improves overall performance as a result. Figure 3 shows improved front-end performance with an EMISSARY cache, on top of FDIP, which exceeds the gains from doubling the size of the L1-I. 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 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 [46]) with a very large entanglement table (>200KB) that consumes 6 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 L1-I cache. EMISSARY policy is the EMISSARY cache at L2 with 8 protected ways per set and 2X IL1 is the configuration with L1-I twice (64KB) the size of the baseline configuration (32KB). As shown in the figure, EIP-Analytical outperforms EMISSARY but still falls short of FEC-Ideal. Furthermore, when EIP and EMISSARY 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.

Figure 4 shows FEC lines of each benchmarks 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 10 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

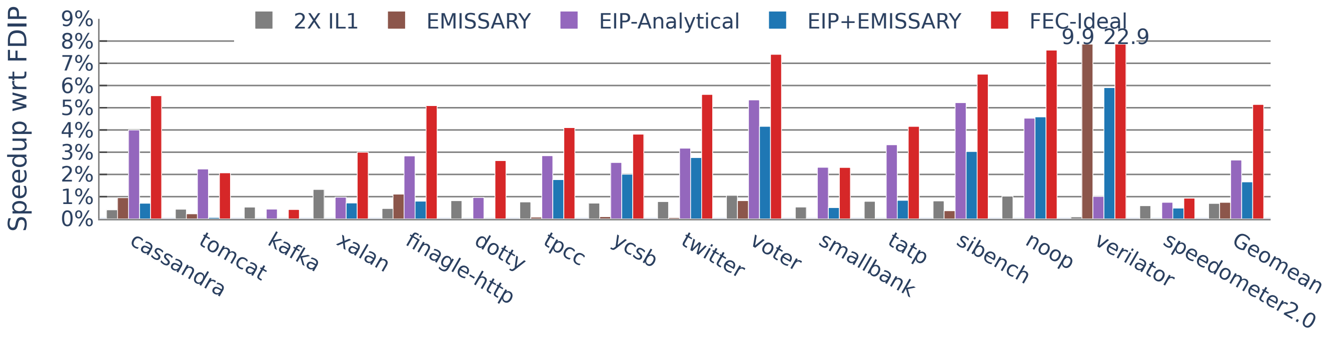

Figure 3. Performance gain of various prior techniques on all benchmarks

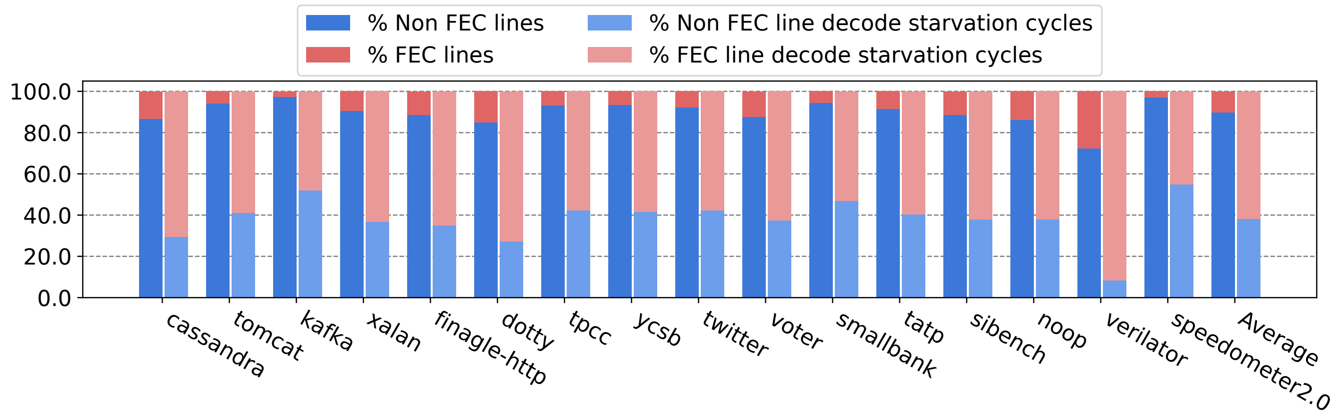

Figure 4. First bar shows the dynamic number of FEC lines as percentage of total lines. Second bar shows the decode starvation cycles caused by FEC w.r.t total decode starvation cycles.

prefetcher on a very small subset of fetched lines, but still achieve most of the available gain.

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 5.

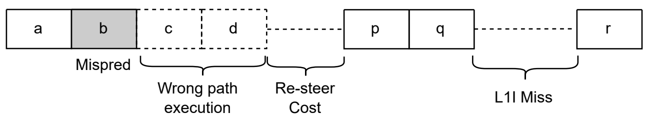

Figure 5. An example showing a sequence of instructions. Each box shown represents one instruction. Dashed boxes are the instructions in the wrong path and dashed line shows wasted cycles due to resteering along with the cost of a miss in instruction cache

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 can't 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 too late to prefetch effectively.

4.1 Selecting Prefetch Candidates

Drawing from the insight that the performance criticality of 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 [17, 18, 46, 51]. Second, by considering prefetch candidates only among FEC lines, PDIP filters out a significant number of unnecessary prefetches, improving its effectiveness.

4.2 Selecting a Trigger Instruction

We will use the example in Figure 6 to illustrate how PDIP finds a trigger and associates it with a prefetch candidate.

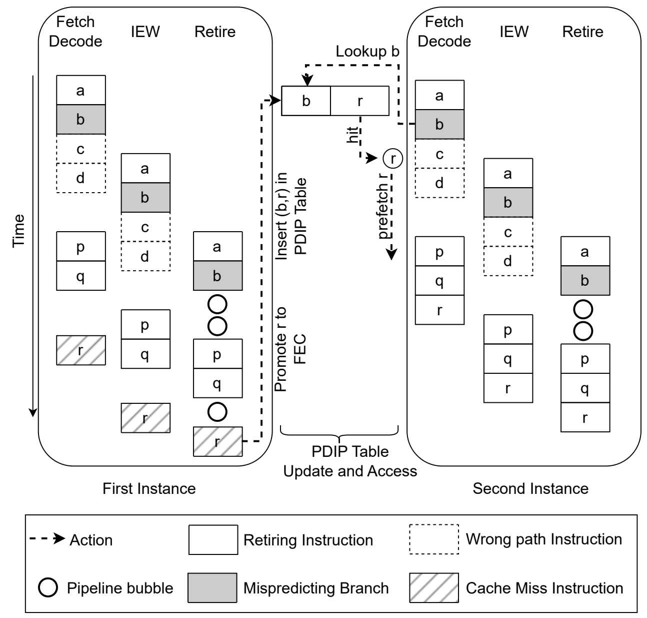

Figure 6. An example showing sequence of instructions in the processor pipeline.

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 [38]), 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 (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 the front-end stall event. There are a couple of categories of events that can expose L1-I misses to front-end stalls. The first are those that 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 L1-I misses that miss in both the L1-I 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 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 block *r* retiring without encountering a bubble in the Second Instance.

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 L1-I 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 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 L1-I. Thus, as FDIP loses effectiveness due to the increased pressure on the branch predictor, the problem that FDIP is trying to solve (L1-I 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 L1-I misses that FDIP can no longer tolerate. And as the code footprint continues to increase, the third cost can begin to dominate.

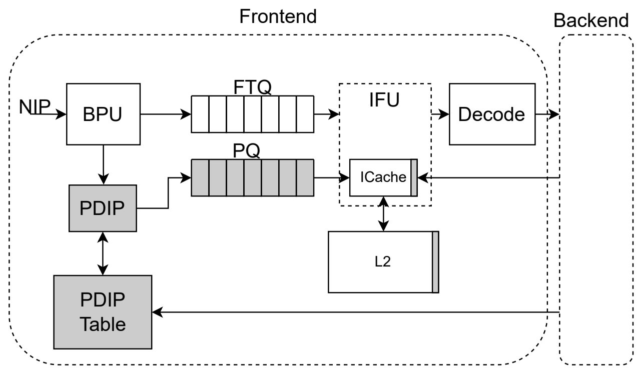

Figure 7. PDIP Pipeline showing new components added in gray blocks

5 Design Implementation

Figure 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 NIP (Next Instruction Pointer) 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.

5.1 PDIP Table

The PDIP table associates a prefetch candidate to 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 to the block address of the trigger instruction instead of the PC address of the trigger. Table insertion or look up, therefore, uses 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 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 8. A set associative table design is used to reduce conflict misses. All configurations of the PDIP table we evaluate use a 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 represents the 3rd and 4th following blocks; thus, when triggered, block r, r+3*blocksize, and r+4*blocksize would be prefetched.

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.

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.

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, but also used by PDIP), in both the L1-I and L2, would be about 4 KB for our configuration. Our default PDIP table configuration has 512 sets, is 8 way set associativity with 2 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 and 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 8 ways.

6 Simulation Methodology

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

6.1 Simulation Model

Our baseline CPU configuration is modeled after a Golden Cove [12] (commercially known as Alder Lake) CPU core microarchitecture using the gem5 [20] simulator. Table 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 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.

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 L1-I. 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 [28] 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)[52]/ ITTAGE(64KB)[50]
BTB size	8K entries (119.01 KB)
FTQ	24 entry [28]
Prefetch Queue	40 cachelines
Decode/Retire	12 wide
ROB Entries	512
Issue/Load/Store Queue	194/ 144 / 112
Int/Vec Registers	448 / 400

Table 1. Processor configurations

6.3 Benchmarks

We use 16 widely used client-side and server-side multi-threaded workloads with large code footprints to evaluate PDIP. Table 2 contains all front-end heavy benchmarks used from various benchmark suites [4, 16, 21, 22, 44]. Benchmarks with an L1-I 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.

6.4 OS and IO bottlenecks : Full System

In a Full System simulation, OS and IO bottlenecks could impact the overall performance and 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. E.g., one trick we deploy is to use "retired instruction counts" rather than

Benchmark Suite	**Benchmarks**
DaCapo [21]	`cassandra` [1], `tomcat` [3], `kafka` [2], `xalan`
Renaissance [44]	`finagle-http` [10] , `dotty` [5]
OLTB Bench [22] (PostgreSQL [7])	`tpcc` [9] , `ycsb` [13], `twitter`, `voter`, `smallbank`, `tatp`, `sibench`, `noop`
Chipyard [16]	`verilator` [11]
Browser Bench [4]	`speedometer2.0` [8]

Table 2. Benchmarks used to evaluate PDIP.

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 variability introduced by IO events we used very low latency for disk IO operations so that a process need not switch out waiting for 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.

6.5 Policies Evaluated

EMISSARY 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 8 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 8 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 L1-I. The main idea with 2X IL1 is understanding tradeoffs between increasing cache size vs. investment in new approaches. The second configuration is EIP [46], a recent instruction prefetching mechanism. That work evaluated EIP using the ChampSim [14] 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 didn't provide any improvement over 40 entries, thus we used a 40-entry history buffer for all configurations of EIP.

Policy Name	**Description**
Baseline	Golden Cove like core
EMISSARY	Priority Ways{L2(8-ways)}
PDIP(S)	PDIP with S KB PDIP Table
EIP-Analytical	Analytical model of EIP [46] with large storage budget for performance
EIP(S)	EIP prefetcher with S KB storage
2X IL1	64KB Instruction Cache

Table 3. Policies Table

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. L1-I miss latencies are captured at fetch but used at commit to compute entangling distances. To avoid misses in the instruction TLB, full physical address is stored in the entangling table. Similar to the PDIP pipeline, if the PQ is full then any new prefetch request is 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 3. We measure performance relative to baseline in Instructions Per Cycle (IPC), and we use Geometric Mean for the mean IPC speedup.

7 Evaluation

We evaluated the benchmarks discussed in Section 6.3 on the policies described in Table 3 using the following metrics: IPC, prefetch accuracy, prefetch coverage and prefetch rate per kilo instructions. We examine multiple PDIP Table sizes.

Figure 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.

7.1 Performance Analysis

Figure 10 shows relative IPC gains across our benchmarks for the policies in Table 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 5 times lesser storage. The PDIP(44) shows a geomean speedup of 3.15% over the FDIP baseline as compared to 1.5% speedup of EIP(46) at similar storage budget.

As shown in Figure 3, EIP suffers when paired with EMISSARY, lacking synergy with the state-of-the-art replacement algorithm. Conversely, PDIP is carefully designed to complement both FDIP and EMISSARY, and provides additional gains over each resulting in a geomean speedup of 3.7%. The combination of PDIP(44)+EMISSARY thus captures 72.5% of FEC-Ideal, described in Section 3.

Since EMISSARY preserves instructions in L2, one drawback is that it causes contention for L2 data accesses. For

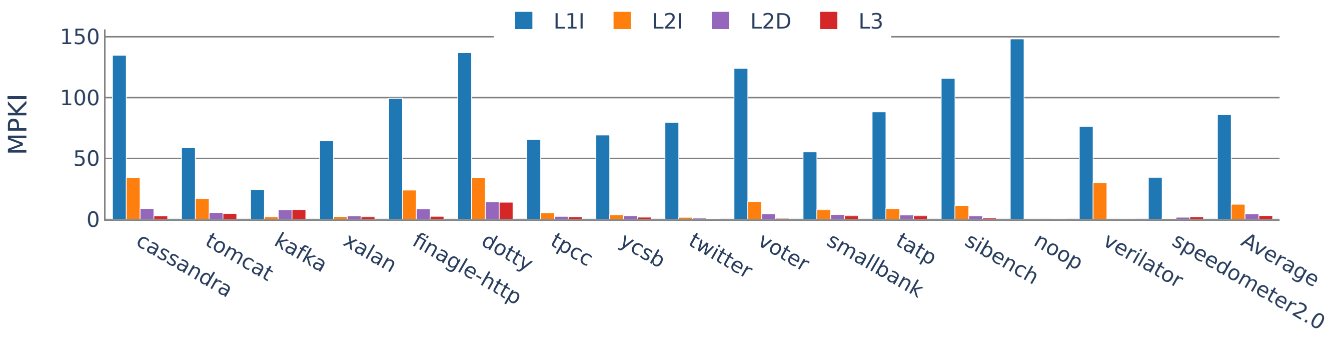

Figure 9. Misses Per Kilo Instructions (MPKI) at L1-I, L2-I and L2-D (instruction and data misses in the L2 cache, respectively), and L3 caches of benchmarks presented in this work

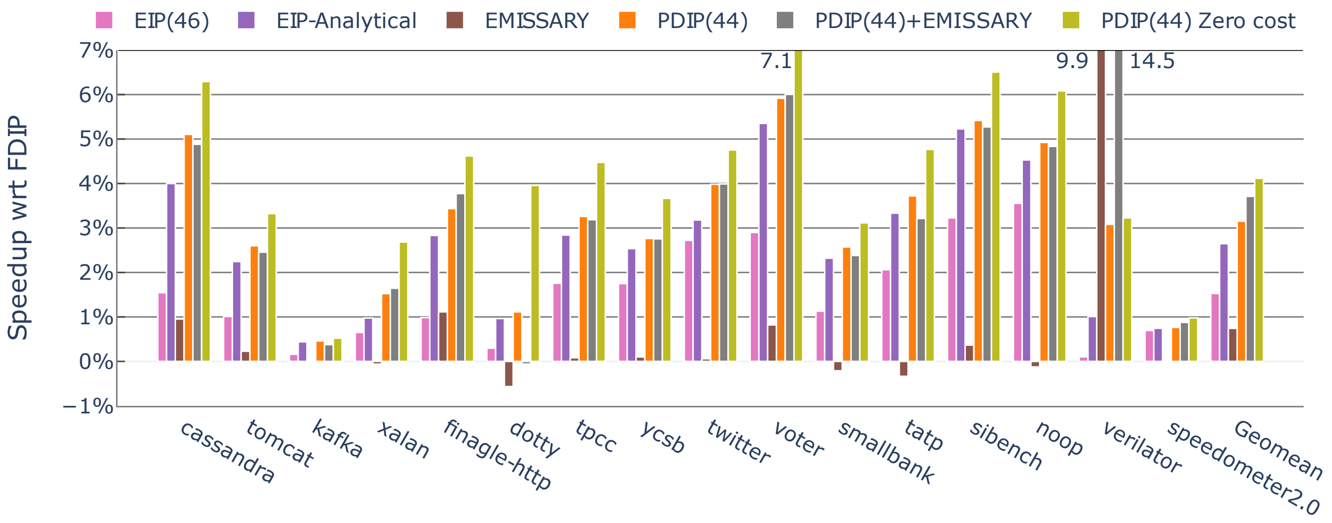

Figure 10. Speedup Comparison

example, the dotty, tatp and smallbank benchmarks all show a considerable increase in L2 data MPKI with EMISSARY 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 enabled. This results in PDIP+EMISSARY having slightly lower performance than PDIP only in such scenarios. On the other hand, for benchmarks like verilator with very low L2 data pressure, they are more complementary.

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 L1-I. 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 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.

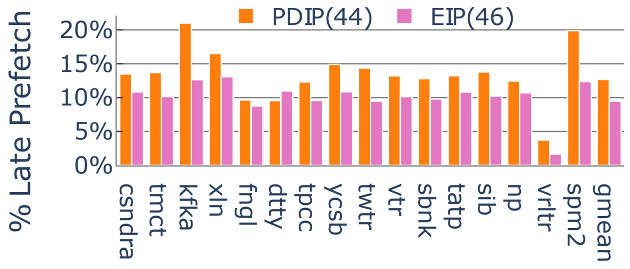

Figure 11. % of Late Prefetches per Benchmark in PDIP(44)

Table 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

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

Table 4. Average Prefetch per Kilo Instructions (PPKI) and Prefetch Accuracy of all prefetch policies

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 accuracy. With large code footprints, too many inaccurate prefetches may evict useful lines in the L1-I 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 accuracy of 55%, while the EIP policies shows 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 similar storage budget. A storage-limited version, PDIP(11), issues the same number of prefetches as EIP(46) while being 4 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 2 times more prefetch requests, putting more pressure on the L1-I with more inaccurate prefetches.

7.3 Prefetch Effectiveness

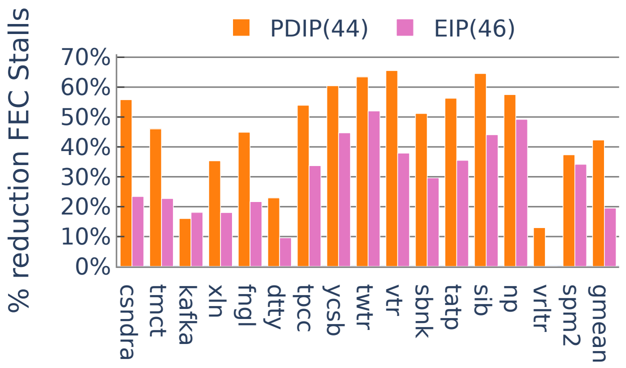

Figure 12. % reduction FEC stalls per benchmark in PDIP(44) and EIP(46)

As discussed in Section 3, L1-I 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 12 shows that PDIP reduces FEC stalls by 50% or more in over 9 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 L1-I pressure and fewer FEC lines (such as `kafka` and `speedometer2.0` as shown in Figures 9 & 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 10 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 work in tandem, reducing FEC stalls by 46% in the PDIP+EMISSARY configuration. Despite the focus on criticality, Section 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, while 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.

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 13 shows performance gain with respect to the FDIP baseline. We store up to 2 targets and 4 consecutive offsets per entry for all PDIP policies, as empirical analysis showed 95% of targets are stored with 2 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` shows either improved or same performance with increase in PDIP Table size. We used optimized (using Facebook's BOLT [41]) binary which has unusually long basic blocks which don't fit in the PDIP Table well. In case of `verilator` increasing mask bits per entry shows better scaling than increasing total number of entries.

7.5 Energy and Area Analysis

We modified McPAT [36] to model the PDIP structures to generate energy and area overheads. Table 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 5. Percentage increase in CPU core Energy consumption and Area over baseline modeled in McPAT

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 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, but also (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 entry, 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 14 shows that at lower 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 uses the same tracking hardware as EMISSARY, PDIP paired with EMISSARY provides the most storage efficient solution at any BTB size, but for brevity, it is omitted from the following discussion.

Figure 15 compares the storage effectiveness of a prefetcher as compared to scaling the BTB. It shows one of the PDIP configurations is always a 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. Furthermore, this also corroborates Ishii et al's[28] observation that prior prefetching techniques provide little performance improvement over modern FDIP machines with large BTBs [12, 47] as evidenced in Figure 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).

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 16 shows that, on average, branch mispredictions contribute to 89% of the issued prefetch targets, while last taken branches contribute to only 11%.

8 Related Work

8.1 Hardware Instruction Prefetchers

EIP [46] 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 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 [28] agree with ours, that EIP does not show heavy improvement over an aggressive FDIP frontend.

FNL+MMA [51] prefetcher combines two techniques – Footprint Next Line(FNL) and Multiple Miss Ahead(MMA) prefetcher. FNL predicts the "worth" of the next 5 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 miss again and similarly a 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 [17, 23, 30, 49] were proposed to prefetch instruction blocks well ahead of time using a history buffer which records the sequence of

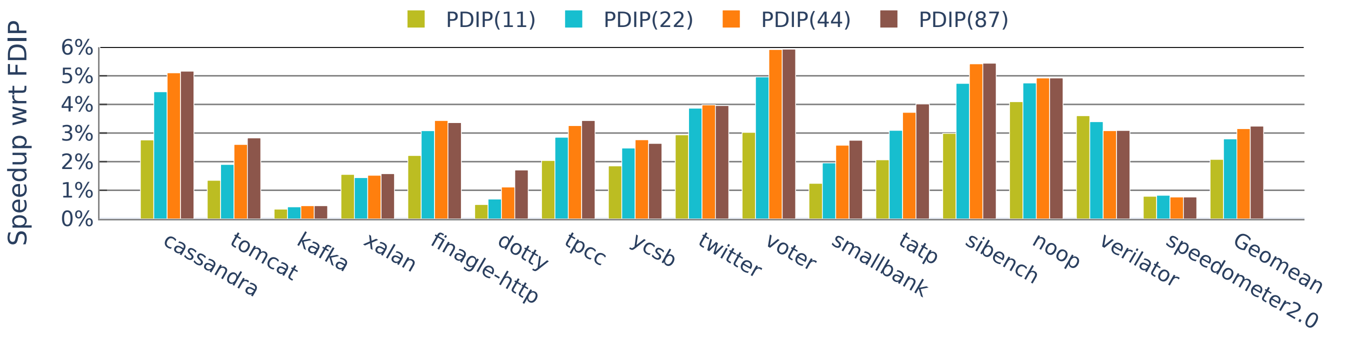

Figure 13. PDIP Policies with various PDIP Table configurations

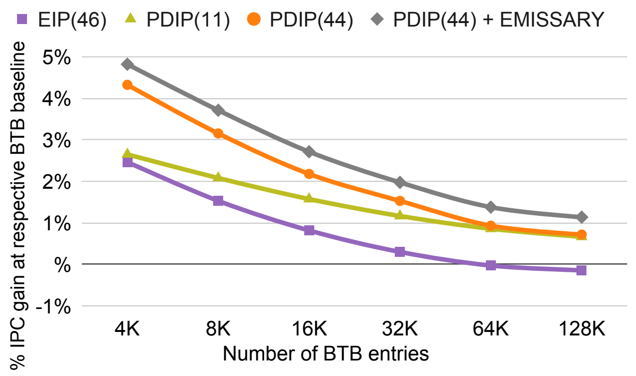

Figure 14. % IPC speedup of prefetch policies at various BTB sizes.

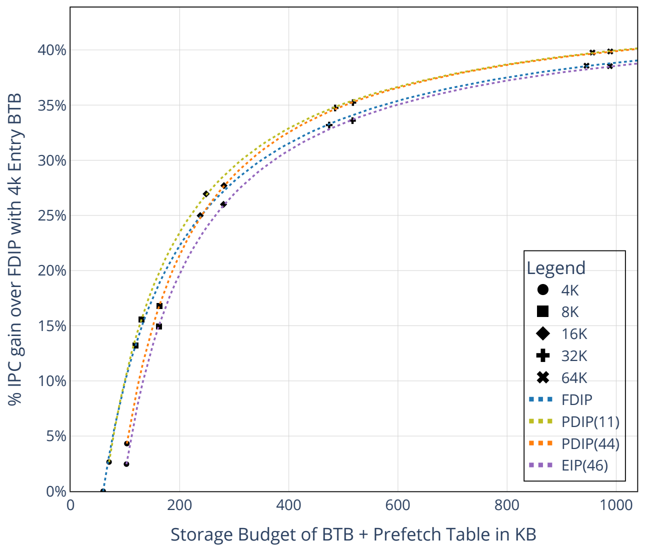

Figure 15. IPC performance gain across different policies at BTB sizes 4K (59KB), 8K (119KB), 16K (237KB), 32K (473KB), and 64K (945KB). PDIP(11), PDIP(44) and EIP(46) needs 10.875KB, 33.5KB and 46KB additional storage respectively.

cache blocks. MANA [17] and PIF [23] (Proactive Instruction Fetch) were similar to the data cache prefetching technique [40]. These techniques take advantage of 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.

Similary SN4L+Dis+BTB [18] combines three techniques. SN4L handles contiguous blocks, Dis handles non-contiguous blocks and BTB is an improvised Confluence[31] 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 [49] 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 L2 cache.

Temporal instruction fetch streaming (TIFS) [24] also works based on record and replay technique. TIFS records the streams

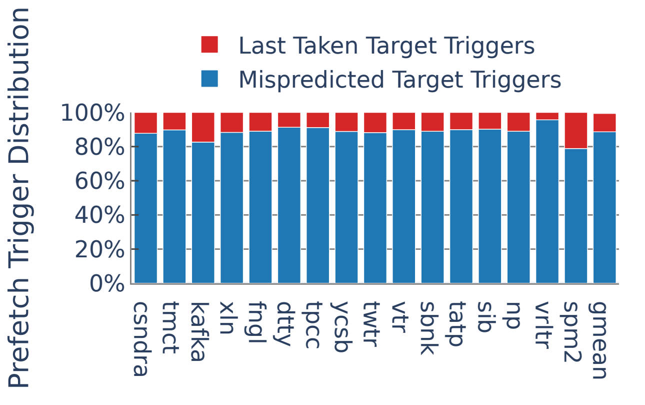

Figure 16. Distribution of prefetches based on Prefetch Trigger scenario

of blocks that miss 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 causing misses repeat. The Temporal Ancestry Prefetcher (TAP) [25] is another prefetcher which 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 harware 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 [33, 39] prefetch lines that missed in the same context last time. RDIP [33] uses return address stack(RAS) as the signature. The key observation is that the misses seen in a given context repeat next time and return address stack or calling context is used to capture the context. D-JOLT [39] is an improved technique which not only uses RAS as context but also captures the blocks which are accessed after long range and short range in a given context so as to send timely prefetch requests. Another key difference in RDIP and D-JOLT is the way 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 number of returns that happened in reaching a given point in the execution.

Branch predictor based prefetching techniques [27, 34] prefetch instruction blocks following a branch using the predicted target. JIP [27] maintains a hierarchy of tables for direct branch with fewer targets and 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 [30] is a storage efficient implementation of history buffers which exploits spatial locality. It is specifically designed for applications running multiple threads which 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 per core separately. Only one core updates the history buffer in LLC.

Prefetching along the wrong path is proposed in [43]. 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 L1-I.

8.2 Software Instruction Prefetchers

Software prefetching techniques [19, 32, 37] typically require changes to the Instruction Set Architecure (ISA) such that instruction lines are prefetched well ahead of their use. These techniques involve inserting prefetch instructions in the code. Cooperative Prefetching [37] technique uses the compiler to automatically identify injection sites using static analysis. AsmDB [19] uses execution profile information to insert prefetches to improve accuracy. I-SPY [32] also uses profile information but encodes context information using a special instruction which 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 technqiues could be used along with PDIP to improve overall performance.

9 Conclusion

Modern high-performance processors incorporate decoupled front end designs (eg, FDIP), enabling aggressive prefetch of instruction cache lines driven by the control flow predictions of the branch predictor. Many of the recent advances in instruction prefetchers in the literature are not designed to work in concert with these front end designs, despite the fact that they significantly change the access stream and miss stream of the Instruction Cache.

This work presents PDIP, a prefetcher carefully designed to complement both a decoupled front-end, but also an Instruction Cache also designed to complement a decoupled front end. This allows it to provide additional gains, even over these aggressive designs, with modest hardware overhead. 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 other SOTA FEC driven policies and achieves upto 72.5% of FEC-Ideal.

Acknowledgments

We thank members of the Liberty Research Group and Intel Labs for their support and feedback on this work. We also thank the anonymous reviewers for the comments and suggestions that made this work stronger. This material is based upon work supported by Intel, and the National Science Foundation under Grant CCF-2107257.

References

[1] Apache cassandra. http://cassandra.apache.org/.
[2] Apache kafka. https://kafka.apache.org/.
[3] Apache tomcat. https://tomcat.apache.org/.

[4] Browserbench. "https://browserbench.org".

[5] Dotty scala compiler. "https://github.com/lampepfl/dotty".

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

[7] Postgresql. "https://www.postgresql.org/".

[8] Speedometer2.0. "https://browserbench.org/Speedometer2.0/".

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

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

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

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

[13] Ycsb. "https://github.com/brianfrankcooper/YCSB/".

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

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

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

[49] 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.

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

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

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

[53] 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.

A Artifact Appendix

A.1 Abstract

This artifact submission corresponds to the Priority Directed Instruction Prefetching (PDIP) technique presented in the paper. PDIP is a novel instruction prefetching approach designed to complement Fetch Directed Instruction Prefetching (FDIP) in modern processors. The primary goal of PDIP is to address front-end bottlenecks caused by instruction cache capacity misses, particularly in scenarios where FDIP struggles. The artifact presents the installation instructions, experiment workflow and reproduction of results of PDIP. It includes gem5 checkpoints for all benchmarks utilized in the experiments, along with the gem5 binary. To facilitate easy execution, a Docker image is provided to setup the environment. The artifact's evaluation metric focuses on IPC speedup over the baseline configuration, calculated from gem5 stats. The results are reported in a provided CSV file, demonstrating the effectiveness of PDIP in achieving a geomean speedup of 3.2% IPC speedup across critical workloads.

A.2 Artifact check-list (meta-information)

- **Program:** Included gem5 checkpoints of all benchmarks utilized. Details of benchmarks in Table 2
- **Binary:** Included gem5 binary and benchmark checkpoints
- **Run-time environment:** Docker image provided to run on Linux. Root access required
- **Hardware:** Machine with > 20 GB RAM.
- **Metrics:** IPC speedup over baseline configuration calculated from gem5 stats is the main metric of comparison reported. Further Gem5 output files (stats.txt, config.json ...) generated for each run
- **Output:** Speedup over Baseline config provided as a csv.
- **Experiments:** Bash script provided to run all benchmarks on each configuration and report speedup over baseline
- **How much disk space required (approximately)?:** 100 GB
- **How much time is needed to prepare workflow (approximately)?:** 15 mins
- **How much time is needed to complete experiments (approximately)?:** 12 hrs
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** 10.5281/zenodo.10553297

A.3 Description

A.3.1 How to access.

- Download Artifact from Zenodo :
  - https://doi.org/10.5281/zenodo.10553297

- Download Size : 40 GB
- Will require approximately 100GB of disk space when extracted

A.3.2 Hardware dependencies.

- A machine with > 20GB memory required.
- A machine with > 100GB disk space required.

A.3.3 Software dependencies.

Linux Operating system with Docker support is required. A docker container is provided with necessary environment setup.

A.4 Installation

Prerequisites. Before you begin, ensure that you have the necessary permissions to use Docker, as it may require sudo access.

Extract Artifact to preferred Directory .

```
tar -xf PDIP-AE.tar.gz -C <project_dir>
```

Setting Up the Environment with Docker.
Navigate to the project's root directory.

```
cd <project_dir>/PDIP-AE/
```

Build Docker

```
sudo bash build_docker.sh
```

Run Docker

```
sudo bash run_docker.sh
```

This will initiate the Docker container setup and configure the environment. `PDIP-AE` directory will be mounted at `/qpoints` within the docker environment.

Basic test.
A basic test has been provided that will invoke the gem5 binary and run one benchmark for 1000 instruction to ascertain if the environment is setup correctly.

```
cd scripts
bash  run_test.sh
```

The test will run gem5 and end with the following message

```
Exiting @ tick <any_value> because a thread
reached the max instruction count
```

A.5 Experiment workflow

All the necessary steps to run experiments and reproduce the results are contained in a single script. This script automates the execution of all 16 benchmarks on gem5, using both the baseline and PDIP(46) configurations.

```
sudo bash run_docker.sh
cd  /qpoints/scripts
bash  run_all_bmk_base_pdip.sh
```

A.6 Evaluation and expected results

The aforementioned script generates all the detailed stats in `PDIP-AE/ae-runs` directory and the key result of speedup of PDIP(46) over baseline (Fig 10) is computed from the results and stored in `PDIP-AE/ae-result.csv`.

Thus a single script runs all experiments and extracts the key results for evaluation. The expected reference results are provided in `PDIP-AE/ae-result-expected.csv`

A.7 Experiment customization

Other Configurations. The base script only runs PDIP(46) but `PDIP-AE/scripts/run_all_cfgs.sh` will run all the configurations mentioned in the paper. This script can be modified to run specific configurations alone. Variables like PDIP_SETS, PDIP_WAYS etc can be modified to run different table sizes or control specific policies of the algorithm.

Other Statistics. As mentioned previously `PDIP-AE/ae-runs` contains all the detailed gem5 stats. Other metrics mentioned in the paper can be inferred from these and new metrics for comparison can also be computed.

Changing Core parameters. All standard gem5 options can also be modified to do further experiments like changing cache sizes, widths and latencies of different stages, choice of branch predictors/prefetchers and other such core parameters. The scripts in `PDIP-AE/scripts/` can be modified or used as an reference to conduct such studies

Other benchmarks. The user can provide their own checkpoints of other benchmarks to validate the results on other varied programs.