UDP: Utility-Driven Fetch Directed Instruction Prefetching

Surim Oh
University of California, Santa Cruz
USA
soh31@ucsc.edu

Mingsheng Xu
University of California, Santa Cruz
USA
mxu61@ucsc.edu

Tanvir Ahmed Khan
Columbia University
USA
tk3070@columbia.edu

Baris Kasikci
University of Washington
USA
baris@cs.washington.edu

Heiner Litz
University of California, Santa Cruz
USA
hlitz@ucsc.edu

***Abstract*—Datacenter applications exhibit large instruction footprints causing significant instruction cache misses and, as a result, frontend stalls. To address this issue, instruction prefetching mechanisms have been proposed, including state-of-the-art techniques such as fetch-directed instruction prefetching. However, our study shows that existing implementations still fall far short of an ideal system with a perfect instruction cache. In particular, up to 588.47% of potential IPC speedup of existing processors hides due to frontend stalls, and these frontend stalls are due to inaccurate and untimely instruction prefetches. We quantify the impact of these individual effects, observing that applications exhibit different characteristics that call for adaptive application-specific optimizations. Based on these insights, we propose two novel mechanisms, UDP and UFTQ, to improve the accuracy of FDIP without negatively affecting timeliness while leveraging prefetches on the wrong path. We evaluate our technique on 10 data center workloads showing a maximal IPC improvement of 16.1% and an average IPC improvement of 3.6%. Our techniques only introduce moderate hardware modifications and a storage cost of 8KB.**

***Index Terms*—Instruction prefetching, frontend stalls, data center.**

I. Introduction

The instruction footprint of modern data center applications significantly exceeds the size of any of today's server's instruction caches (icaches). To make matters worse, Google and Facebook have reported that their code sizes increase at a rate of up to 20% per year while processor caches have experienced only moderate gains in size over the last decade. These footprints cause frequent icache misses leading to frontend stalls, as the CPU pipeline is waiting for the next instructions to be fetched from memory. Google has reported that processors in their fleet spend almost a quarter of their cycles on these misses [16], resulting in inefficiencies worth millions of dollars while causing significant carbon emissions.

To address this challenge, prior work has explored instruction prefetching mechanisms. Reinman has introduced Fetch Directed Prefetching (FDIP) [47], a hardware prefetching technique that exploits existing resources such as branch predictors as prefetching engines. FDIP decouples the branch prediction from the instruction fetch. By predicting future instructions faster, the branch prediction unit can run ahead of the fetch engine while emitting prefetches to hide the memory access latency of instruction fetches. FDIP is widely adopted by the industry and implemented on commercial processors [44], [50], [57] as it introduces only small hardware overheads while providing high prefetching accuracy as long as the branch predictor performs well.

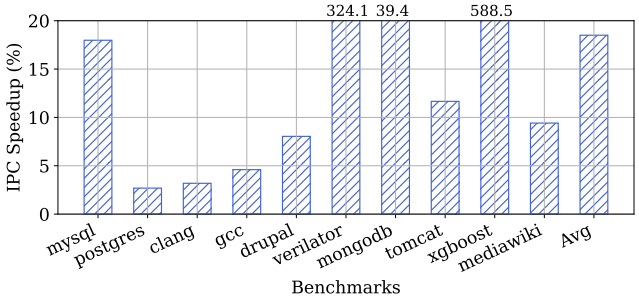

Fig. 1. Ideal icache speedup over today's FDIP baseline. The ideal speedup is in the range of 2.69% to 588.47%.

Unfortunately, large code-footprint applications generally also contain many branches exceeding the capacity of the branch target buffer (BTB). BTB misses (and branch mispredictions) limit the accuracy of FDIP, as after suffering from a BTB miss, the frontend will be on the wrong path, prefetching unuseful instruction cache lines while polluting the icache. To quantify this problem, Figure 1 compares the performance improvement of a perfect icache (every access is a hit) against the state-of-the-art FDIP baseline. As can be seen, for most of the data center applications, the perfect icache provides a speedup of 2.69% to 39%, while for `xgboost` and `verilator`, the IPC improvement is up to 7× and 4×.

Prior work, including Boomerang [38], Shotgun [37], and Confluence [30] try to address this challenge by improving

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00089

the hit rate of the BTB. All three works introduce significant hardware complexity and storage overheads and, hence, have not been adopted in existing processor designs. They also do not address the problem of branch mispredictions and hence do not address the root cause of the problem: executing unuseful prefetches on the wrong path. To the best of our knowledge, Ishii [28] represents the state-of-the-art FDIP implementation deployed by today's processors. It features a large BTB, partly solving the BTB capacity issue while introducing feasible to implement techniques such as taken-only branch history and post-fetch correction. Another line of work has proposed to utilize software prefetching [16], [31], [32], however, these techniques require profiling and re-compilation and cannot adapt to dynamic workload changes.

We propose a different approach for addressing the existing frontend bottleneck for data center applications. Towards this goal, we investigate the state-of-the-art FDIP implementation to understand why it falls short of the ideal performance provided by a perfect icache. In particular, we analyze whether FDIP mainly suffers from inaccurate wrong-path prefetches (polluting icaches) or from untimely late-prefetches. These two causes are intrinsically connected, as for prefetch timeliness, the frontend needs to run ahead as far as possible, while for ensuring prefetching accuracy, it is beneficial to limit the number of prefetches emitted on the wrong-path. Our analysis determines the usefulness of prefetches both on the on-path and off-path and explores how the frequency of branch mispredictions and their recovery affect the behavior of FDIP. Our analysis reveals that existing techniques fail to ensure timeliness while emitting many inaccurate wrong-path prefetches. Wrong-path prefetches frequently pollute the icache, jeopardizing the effect of FDIP whose aim is to reduce icache misses.

To address this challenge, we propose two new mechanisms for improving the efficacy of FDIP. First, we introduce UFTQ, which adapts the runahead distance of FDIP dynamically and in an application-specific way by configuring the depth of the fetch target queue (FTQ). We show that tuning the FTQ depth dynamically can improve the timeliness and usefulness of prefetches providing an IPC improvement of up to 37.2% and of 4.9% on average.

To further improve the efficiency of FDIP, we introduce UDP, which increases performance further over UFTQ by learning the utility of individual prefetch candidates. We propose an efficient implementation to learn the utility of prefetch candidates proposed by FDIP, including useful off-path prefetches. In particular, our technique learns useful off-path prefetches that correspond to code locations after the merge point[1], however, which are emitted before a branch misprediction is resolved. As a result, UDP can almost perfectly eliminate unuseful prefetches while leveraging a deep FTQ for good timeliness. UDP provides an IPC uplift of up to 16.1% and of 3.6% in average. In summary, we make the following

[1]Merge point: The instruction after which two control-flow paths merge, for instance, the first instruction after an `if{} else{}` block

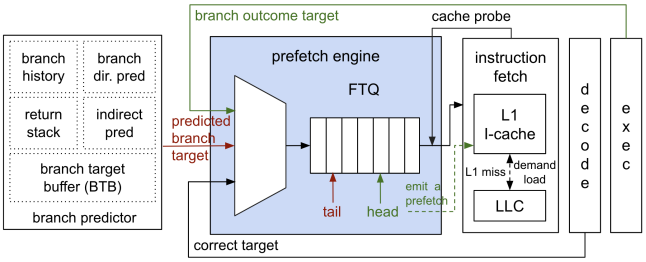

Fig. 2. Fetch Directed Instruction Prefetching. The decoupled frontend inserts fetch-blocks at the tail while FDIP emits prefetches at the head. The backend takes the instructions from the FTQ.

contributions:

- A detailed analysis of the state-of-the-art FDIP implementation revealing why its performance falls significantly short of a perfect icache
- UFTQ, a mechanism that dynamically adapts the FTQ size based on workload characteristics
- UDP, a technique that prefetches only useful instructions both on the on-path and off-path
- A comprehensive evaluation of UDP for 10 large instruction footprint applications showing an IPC performance improvement of 3.6%.

UDP is built on top of an open-source Scarab [9], [10] simulator and incorporated into the open-source Scarab infrastructure [11]. Scarab+UDP is available at *https://github.com/Litz-Lab/scarab* and the tool is available at *https://github.com/Litz-Lab/Scarab-infra*.

II. Background

Fetch Directed Instruction Prefetching (FDIP) [47] is a hardware prefetching technique that leverages the branch predictor to prefetch instructions. The technique is based on the insight that the branch predictor can advance instructions faster than the backend. In particular, a decoupled frontend can process one (or more) fetch blocks per cycle while the backend can only execute a fixed number of instructions per cycle. A fetch block is defined as a fixed-sized, aligned number of instruction bytes. Modern architectures support fetch blocks of a size of 32 Bytes [28] and can look up all instructions of a block in the (banked) BTB simultaneously, to extract control flow changing instructions. For each branch within the block, the predictor is consulted to determine whether the branch is taken or not-taken. A taken branch terminates the fetch block and finishes processing of said block in the given cycle. The decoupled front end then inserts the start and end address of the fetch block into a data structure referred to as the fetch target queue (FTQ).

The FTQ is utilized for two purposes. First, it informs the instruction fetch engine and decoder about block boundaries so that it can retrieve sequential instructions from the icache until terminated by a taken branch. Second, the FTQ enables FDIP to reduce icache misses.

FDIP scans the FTQ to look up fetch-blocks in the icache before the icache will access them. FDIP can run ahead of

the backend as it can look up one block in the icache per cycle, whereas the backend can decode only, for instance, six instructions per cycle (depending on the width of the processor). The icache lookups generated by FDIP generate prefetches, reducing the access time of later demand accesses by the fetch/decode stage. FDIP is effective as it leverages today's highly accurate branch predictors, such as TAGE-SC-L [52] for generating prefetch candidates. An overview of a modern processor architecture featuring a decoupled frontend and FDIP is given in Fig. 2.

One challenge of FDIP is that it relies on the BTB for finding the next branch instructions. On a BTB miss, FDIP will not be aware of the miss and just assume to be processing a large basic block (BBL)[2], issuing likely unuseful wrong-path prefetches causing icache pollution. Branch mispredictions will have similar effects and, as a result, FDIP moves the frontend bottleneck of large footprint applications from the icache to the branch predictor and BTB. Prior work has addressed this challenge by reducing BTB misses through different techniques such as eliminating undetected BTB misses (Boomerang [38]), increasing BTB utilization (Shotgun [37]), prefetching BTB entries (Confluence [30], and increasing storage efficiency (Pdede [55]). These works, however, provide significantly less than ideal performance and, furthermore, introduce complex hardware modifications and storage costs.

Ishii [28] addresses the issue of wrong-path prefetches after a BTB miss through post-fetch correction. This technique detects BTB misses as soon as an instruction is decoded and then immediately flushes the FTQ instead of waiting until the undetected branch is resolved in the execute stage. While this reduces the time FDIP is on the wrong path, we still find that a decoupled frontend that runs far ahead has plenty of opportunities to issue unuseful prefetches.

In the next section, we provide a detailed performance analysis of the state-of-the-art FDIP implementation (Ishii [28]) to determine the root cause of why FDIP falls short of ideal performance. We then introduce UFTQ and UDP, two mechanisms to reduce frontend stalls while only introducing moderate hardware modifications. Both techniques are orthogonal to the prior works in that they can be combined with techniques that, for instance, improve BTB storage capacity to deliver even higher performance gains.

III. Analysis

In this section, we analyze the performance of FDIP for 10 data center applications with large code and branch footprints. We study the optimal hardware parameters for FDIP to optimize performance including the run-ahead distance for FDIP, different throttling mechanisms, the distribution of on-path and wrong-path prefetches, as well as the usefulness of prefetches. Our analysis reveals *why* existing techniques fall short of optimal performance and provides valuable insights on how to address this problem.

[2]A BBL is a sequence of instructions started with the target of a control flow changing instruction (branch, call, jump, return) and ending with a control flow changing instruction.

TABLE I
Studied data center applications. FB: Frontend-Boundness

Applications	FB	Workloads
`MySQL` [7]	60%	`sysbench` OLTP-like database benchmarks [4]
`PostgreSQL` [8]	45%	`sysbench` OLTP-like database benchmarks [4]
`Clang` [2]	35%	Building `SPEC CPU 2017` benchmarks [12]
`GCC`	29%	Building `SPEC CPU 2017` benchmarks [12]
`drupal`	39%	A PHP content management system from HHVM OSS-perf benchmark [58]
`Verilator` [13]	66%	`Verilog/SystemVerilog` simulator
`MongoDB` [17]	62%	`mongo-perf` benchmarks [3]
`tomcat` [1]	38%	Apache's implementation of the Java Servlet and WebSocket
`XGBoost` [60]	87%	Gradient Boosting Library decision tree prediction
`mediawiki` [59]	32%	An open-source Wiki engine

A. Experimental Methodology

Data center applications. Datacenter workloads exhibit code footprints in the order of megabytes, substantially exceeding the size of modern servers' instruction caches. Google has reported [16] that the servers in their fleet spend almost a quarter of their cycles on frontend stalls while web-serving and caching workloads deployed at Facebook are up to 36% frontend-bound [56]. As these workloads are proprietary, we evaluate FDIP for 10 open-source applications that show representative characteristics in terms of frontend-boundness. We briefly describe these workloads in Table I. We set up `sysbench` for `MySQL` and `postgres` with 1M OLTP events on 10 tables. `Clang` and `GCC` builds `538.imagick_r` SPEC2017 benchmark. `mongo-perf` sends complex insert requests to `MongoDB`. `Verilator` is configured to simulate a Boom RiSC-V core running Drysthone. For `XGBoost`, we evaluate the inference phase. Therefore, we first train a decision tree model with the HIGGS dataset [5], maximum depth of a tree of 8, step size shrinkage 1, regression with squared loss, and an evaluation metric of root mean square error resulting in an MB-sized model with 10's of thousands of branches. We observe that almost most of the instructions for `XGBoost` are executed only once.

Simulation Environment. We leverage the open-source, cycle-accurate Scarab [9], [10] simulator to perform our analysis. Scarab supports state-of-the-art branch predictors such as TAGE-SC-L [52] and MTage [53] as well as recent data prefetchers [6], [27], diversified functional units, and a detailed cache and DRAM model leveraging Ramulator [35]. We have extended Scarab with a realistic implementation of a decoupled frontend [46] that supports wrong-path execution and icache prefetching via FDIP [47]. Scarab supports both execution-driven and trace-based frontends. In the execution-driven mode, Scarab leverages Intel's PIN [40] to resteer the binary onto the wrong path, generating accurate off-path instructions. To simulate wrong-path execution for the trace-based frontend, Scarab captures every executed PC and associated instruction to replay instructions on the wrong path. Only if a given wrong-path PC has never been seen before does Scarab generate a NOP. We find that in 99% of the cases, the correct actual instructions are being replayed after warming up the simulator with 50M instructions. As a result,

TABLE II
SIMULATED SYSTEM

Parameter	Value
CPU	Sunny-Cove-like
All-core turbo frequency	3.0 GHz
Frontend width and retirement	6-way
Functional Units	4 ALU, 2 Load, 2 Store
Branch Predictor	TAGE-SC-L [52]
Branch Target Buffer (BTB)	8K entries
Indirect Branch Target Buffer	2K entries
ROB	352 entries
Reservation Station	125 entries (unified)
Data Prefetcher	Stream
Instruction Prefetcher	FDIP [47]
Load Buffer	64 entries
Store Buffer	64 entries
Uncore	
L1 instruction cache	32 KiB, 8-way
L1 data cache	48 KiB, 12-way
L2 unified cache	512 KiB, 8-way
LLC unified cache	Shared 2 MiB/core, 16-way
L1 D-cache latency	4 cycles
L1 I-cache latency	3 cycles
L2 latency	13 cycles
LLC latency	36 cycles
Memory	DDR4-2400 (1 channel)
Decoupled Frontend	
FTQ blocks per cycle	2
FTQ block size	32 B

all aspects, including data dependencies, icache pollution, and multiple consecutive mispredictions, are correctly modeled in trace mode, even on the wrong-path. The only inaccuracy of this approach is that replayed load/store instructions will reuse prior addresses, affecting wrong-path dcache pollution. We analyzed this effect to be minimal (¡1% IPC mismatch) by comparing the same code leveraging the execution driven respectively trace-based frontend. We leverage the execution-driven frontend for all applications where possible and utilize traces to simulate complex, multi-process, and Java-VM-based applications that are not supported by Scarab otherwise. The traces are collected with DynamoRIO [19] and Intel PT [24] containing a precise continuous sequence of dynamically executed basic blocks and memory addresses. We analyze the traces to extract representative steady-state regions.

We configure Scarab to match recent Intel/AMD processors in terms of width and depth summarized by Table II. Note that we will use the same set of applications and simulator configuration in the evaluation presented in Section V.

B. Optimal run-ahead distance (FTQ depth)

The runahead distance of FDIP determines both the timeliness of prefetches as well as the likelihood of emitting wrong-path prefetches. In particular, a frontend that executes many BBLs ahead can emit timely instruction prefetches that reach the icache before being consumed by the backend. On the other hand, a large run-ahead distance increases the likelihood of icache pollution as FDIP spends more time on the wrong path after a branch misprediction. The optimal runahead distance of an application depends on several application properties such as its footprint and IPC performance. We analyze the optimal per-application runahead distance by sweeping the depth of the FTQ.

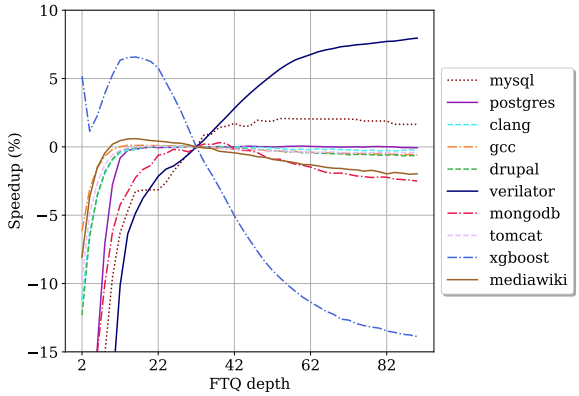

Fig. 3. Optimal FTQ Size : the optimal FTQ size for each application varies from 16 (`xgboost`) to 90 (`verilator`).

Fig. 3 shows the IPC speedup over the state-of-the-art FDIP baseline with an FTQ depth of 32 [28] for the evaluated applications. As we can see, the optimal FTQ depth for each application varies substantially, ranging from 16 to 90. Two applications particularly stand out. `verilator` scales well with large FTQs as it exhibits an extremely large instruction footprint but low branch misprediction rates. As a result, it can run far ahead and thus benefit from improved prefetching timeliness. XGBoost implements large decision tree models. The MB-sized generated source code essentially only consists of conditional statements compiled into a sea of branches. The code exhibits little reuse, high branch misprediction, and BTB miss rates. As a result, XGBoost spends 90% on the off-path, and hence aggressive prefetching leads to substantial cache pollution.

Observation: Applications show variable optimal runahead distance configurations for FDIP.
Insight: *Adapting the FTQ size in an application specific way is beneficial, however, we need to understand the root cause of such performance differences.*

C. Timeliness of Prefetches

The main goal of increasing FTQ depth is to enable FDIP to run ahead further, generating more timely prefetches. This reduces frontend stalls as the fetch stage suffers fewer long latency icache misses. To evaluate this effect, we measure the ratio of demand loads hitting the icache respectively the fill-buffer or miss status hold registers (MSHR). In the case of a timely prefetch, demand loads will retrieve instructions from the icache, whereas if the prefetch was untimely, the demand load will be merged with the prefetch in the fill buffer. Figure 4 shows the $icache_{hits}/(icache_{hits} + MSHR_{hits})$ ratio for the analyzed applications. It shows that except for `xgboost` and `verilator`, the workloads can achieve prefetch timeliness with a small FTQ. `xgboost` and `verilator` require substantially larger FTQs to achieve timeliness.

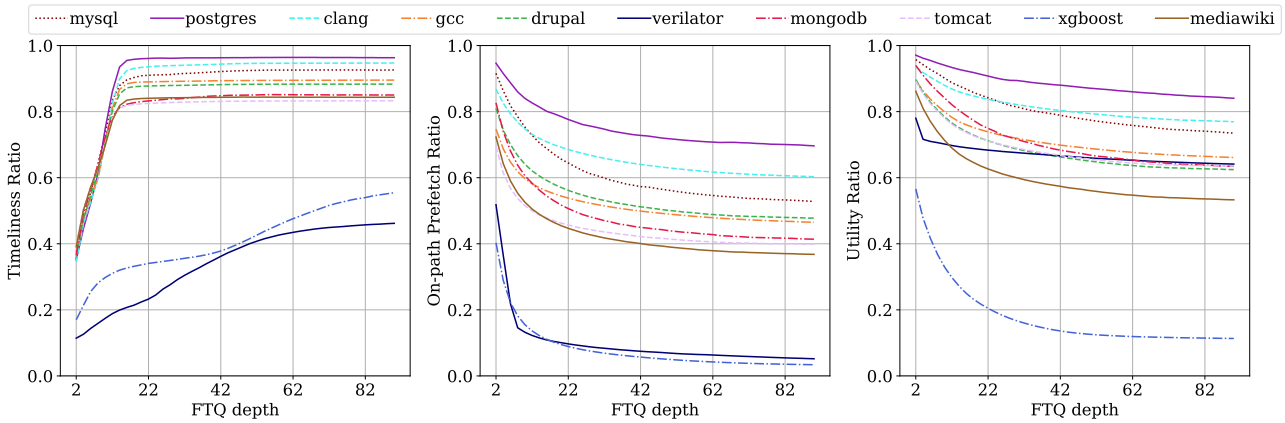

Fig. 4. Timeliness Ratio: Only `xgboost` and `verilator` show improving timeliness with a larger FTQ.

Fig. 5. Onpath Prefetch Ratio: The workloads show a different range of on-path ratio. For example, while `xgboost` and `verilator` mostly go off path as FTQ size increases, `postgres` stays 70% on the right path.

Fig. 6. Utility Ratio: Ratio of useful prefetches. Utility Ratio = Useful / (Useful + Unuseful). While `xgboost` emits more unuseful prefetches with a deeper FTQ, `verilator` emits 70% of useful prefetches.

Observation: Applications require different FTQ depths to achieve timeliness.
Insight: *FTQ size should be configured in an application-specific way.*

D. On-path vs. Off-path Prefetching

As shown in Section III-B, IPC does not scale monotonically with larger FTQ depths. One potential reason for the slowdown of larger FTQ is the increased probability of issuing unuseful prefetches on the wrong-path after a branch misprediction or BTB-miss induced resteer. In Figure 5 we plot the ratio of $on\text{-}path/(on\text{-}path + off\text{-}path)$ prefetches emitted by FDIP. As can be seen, FDIP emits an increasing number of off-path prefetches as the FTQ size increases, potentially polluting the icache.

Observation: All applications show an increasing number of off-path prefetches with larger FTQ depths. There does not exist a one-size-fits-all best FTQ configuration for all applications
Insight: *A new FDIP tuning mechanism is required to enable both accurate and timely prefetches.*

E. Usefulness of Off-path Prefetches

As shown in the previous section, the number of off-path prefetches increases with the FTQ depth. However, off-path prefetches can be useful if they bring in cache lines that are also touched on the on-path. Off-path prefetches are emitted before a mispredicted branch is resolved and hence can be more timely than later on-path prefetches for the same line. Useful off-path prefetches frequently occur before a *merge-point*. To define merge point, we consider the code snippet in Figure 7 and `cond = false`. If the branch in line 1 is mispredicted, FDIP will not prefetch the code for line 2, however, it will prefetch the code for line 3 after the merge point of the misprediction is recovered at a later point in time. Such an off-path prefetch will be more timely than the on-path prefetch emitted after the misprediction is resolved.

```
1 if (cond) a++;
2 else a--;
3 b += a; //Merge Point
```

Fig. 7. Merge Point after a Branch

To provide insight into the usefulness of wrong-path prefetches, we analyze the ratio of useful to unuseful prefetches independently of whether a prefetch was emitted on-path or off-path. A useful prefetch is defined as a prefetch that is hit by an on-path demand load, either in the icache or fill-buffer (MSHR). An unuseful prefetch is defined as a line that is evicted from the icache without being accessed by a demand load. Figure 6 shows the prefetch usefulness for different FTQ depths. By comparing Figure 5 and Figure 6 we can infer the following application characteristics and divide workloads into three categories: off-path prefetches are either 1) very useful (`verilator`), 2) somewhat useful or harmful, and 3) harmful (`xgboost`, `mongodb`). As an example of 1), when considering the larger FTQ and `verilator`, FDIP emits approximately 90% of off-path prefetches as depicted in Figure 5 while the utility ratio is 0.6 in Figure 6. This means that 60% of on/off prefetches are still useful although 90% of the prefetches are emitted on off-path. `xgboost` is an example of 3) and exhibits a utility ratio of 0.1, indicating that the majority of the off-path prefetches are not useful and discarded from the icache without being accessed by a demand load. For some applications (e.g., `mongodb`), FDIP cannot run ahead enough and emit more prefetches with a larger FTQ because of the frequent resteers. Depending on resteer frequency and instruction footprint size, off-path usefulness varies where it can be more or less useful or detrimental. For example, FDIP does not emit more prefetches with a larger FTQ for `mongodb` as 40% of prefetches are on-path, and this

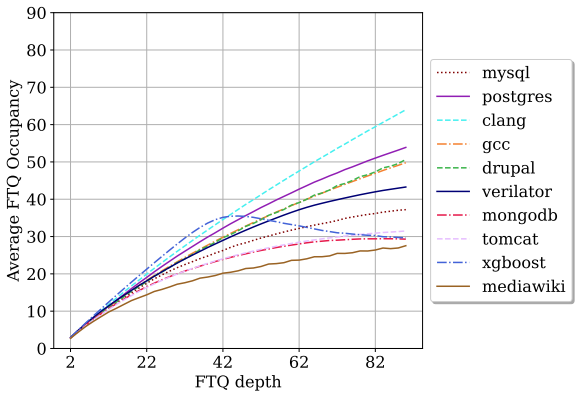

Fig. 8. FTQ occupancy

leads 70% of useful prefetches.

Observation: Some applications benefit substantially from off-path prefetching whereas other workloads suffer.
Insight: *Limiting off-path prefetches through bandwidth throttling and FTQ depth is insufficient. We need to learn the usefulness of both on-path and off-path prefetches.*

F. FDIP Recovery Frequency

As shown in Figure 3, for some applications, the IPC plateaus even for large FTQs as these workloads do not seem to suffer from unuseful prefetches polluting the icache. This can be explained by the following behavior. All branch mispredictions and BTB-misses which steer FDIP onto the off-path are eventually detected and resolved causing a flush of the FTQ. If the recovery frequency is high enough, the FTQ will rarely fill up as this requires **N** cycles, where N represents the depth of the FTQ (This assumes that the decoupled frontend can generate one block per cycle). Recoveries hence act as a natural throttling condition for FDIP. To analyze this effect Figure 8 plots the average FTQ occupancy for the evaluated applications over different FTQ sizes. A line with slope 1 means that there is a low resteer frequency enabling the decoupled frontend to run ahead and fill the FTQ. For instance, this is the case for `clang` which can run far ahead and hence emit a lot of prefetches without being throttled by early resteers. The average occupancy of `xgboost` starts going down with a deeper FTQ. This is because it emits too many unuseful prefetches, leading more time to be spent on the off-path due to icache pollution. `verilator`, on the other hand, does not suffer from too many or harmful off-path prefetches (see Figure 5) because it flushes the FTQ relatively frequently or because the off-path prefetches are useful.

G. Analysis Summary

Our analysis provided the following insights. First, there exist substantial opportunities to improve performance by optimizing FDIP. Second, applications exhibit different performance characteristics, and hence an efficient mechanism has to be able to adapt to these properties. Third, it is insufficient to just focus on on-path prefetches as off-path prefetches can provide more timely prefetches in certain scenarios. Based on these insights, the next Section will introduce two techniques to improve the performance of FDIP.

TABLE III
UTILITY & TIMELINESS RATIO FOR OPTIMAL FTQ SIZE

Application	Optimal FTQ	Utility	Timeliness
`mysql`	56	0.77	0.93
`postgres`	76	0.85	0.96
`clang`	54	0.79	0.95
`gcc`	60	0.72	0.93
`drupal`	28	0.64	0.85
`verilator`	84	0.64	0.46
`mongodb`	38	0.69	0.85
`tomcat`	24	0.69	0.82
`xgboost`	12	0.30	0.31
`mediawiki`	18	0.62	0.83
Average (Geomean)	42	0.65	0.75
Correl. Coefficient	-	0.63	0.21

IV. DESIGN

This section introduces UFTQ and UDP, two novel techniques to increase the performance of the state-of-the-art FDIP mechanism. UFTQ leverages insights from Section III to adapt the FTQ depth based on dynamically learned application properties. This technique aims to find the optimal FTQ depth for a given workload considering both the timeliness and usefulness of prefetches. The UDP technique further improves on UFTQ by learning prefetch-specific utility values. In particular, it learns whether a specific prefetch candidate should be emitted or dropped, regardless of whether it resides on the on-path or off-path.

A. UFTQ: Application-specific FTQ size

In Section III-B, we showed that applications exhibit different optimal FTQ sizes. To leverage this insight, we propose a technique that dynamically adjusts the FTQ size to optimize IPC by maximizing timeliness while limiting icache pollution. Unfortunately, every application exhibits a different IPC and the ”optimal” IPC is unknown for a given workload. To address this problem, UFTQ instead monitors the timeliness and utility of emitted prefetches and then adapts the FTQ size accordingly. Table III shows for each application the optimal FTQ size obtained using exhaustive exploration. As can be seen, the optimal FTQ-len correlates with the utility ratio (useful/unuseful prefetches) and timeliness (icache hit/MSHR hit). Our approach leverages this fact by measuring utility and timeliness at runtime and then computing the FTQ length based on these measurements. We propose three techniques. UFTQ-AUR computes the FTQ length based on utility only, UFTQ-ATR computes the FTQ length based on timeliness only, and UFTQ-ATR-AUR computes the FTQ length considering both timeliness and utility. The techniques initialize the FTQ size to 32 and then measure the utility (and/or timeliness) ratio of the next 1000 prefetches. If the utility ratio is higher than the AUR, our approach extends the size of the FTQ. If the current utility ratio is smaller than the AUR, UFTQ reduces the size of the FTQ.

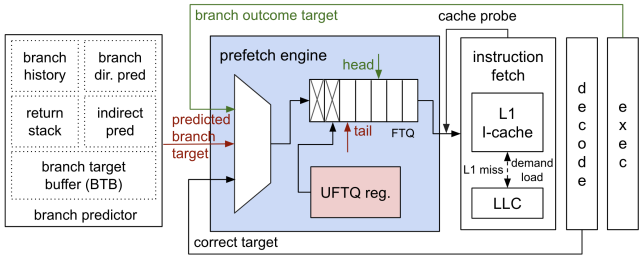

Fig. 9. UFTQ-ATR-AUR Microarchitecture

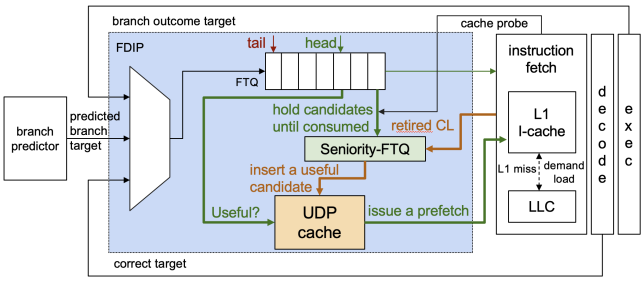

Fig. 10. UDP Microarchitecture

Similarly, UFTQ-ATR utilizes the timeliness of prefetches to adapt the FTQ size. We consider prefetches as timely if they hit the icache and untimely if they hit a prefetch in the fill buffer. We again compute the average timeliness ratio (ATR) to adapt FTQ size and then utilize the same approach as above while replacing AUR with ATR.

The third alternative, UFTQ-ATR-AUR, combines the two techniques. Therefore, we initialize the FTQ with a size of 32 and then perform utility measurements to find the queue depth satisfying AUR. We refer to that queue depth value as QDAUR. After finding QDAUR, we increase or decrease the queue depth until we satisfy ATR. This queue depth is referred to QDATR. We then determine the final FTQ size by computing $FTQ_{size} = -0.34 * QDAUR + 0.64 * QDATR + 0.008 * QDAUR^2 + 0.01 * QDATR^2 - 0.008 * QDAUR * QDATR$ where the equation is obtained by polynomial regression from trained randomly chosen 80% simpoints for memtraced workloads and warm-up phases (50M) for PT workloads. The technique is always-on to adapt to future application phase changes that may alter the ATR or AUR. The maximum queue size is bounded by the physical size of the FTQ, for instance, 64. We assume that the icache holds a prefetch bit for each cache line, which is set when a prefetched line is inserted in the icache. The prefetch bit is cleared when the line is accessed by a demand load. Similarly, entries in the fill buffer contain information about whether they were installed by a prefetch or demand load. We do not consider those bits as a UFTQ-specific overhead as most architectures already implement these features.

The UFTQ technique introduces small hardware overheads. To compute the running AUR and ATR, four 10-bit counters and two 32-bit fixed point registers storing the ratios are required. A simple state machine is needed to adapt the FTQ size based on AUR and ATR. In Section V we evaluate the three techniques UFTQ-ATR, UFTQ-AUR, and UFTQ-ATR-AUR and compare them against the state-of-the-art baseline and an optimal upper-bound mechanism. The proposed UFTQ-ATR-AUR design is shown in Figure 9.

B. UDP: Utility-driven Instruction Prefetch

The presented UFTQ technique is based on two assumptions. The first assumption is that ATR and AUR, respectively, their combination, are accurate proxies for IPC performance. The second assumption is that a single FTQ size is optimal for a given phase to throttle or admit prefetches. While Section III shows that ATR and AUR are good proxies to adapt the FTQ size, an even more effective technique must consider individual prefetch candidates and learn their behavior. We define a prefetch candidate as a prefetch of a cache line that refers to one of the blocks in the FTQ and which is currently not present in the icache. The UDP technique leverages the following two insights. First, on-path prefetching candidates are always useful as they are guaranteed to be utilized by subsequent demand load. Second, off-path prefetching candidates can be useful if they are emitted before an on-path candidate to the same line, in which case they improve prefetch timeliness. UDP utilizes these insights as follows. For each branch predicted by the decoupled frontend, we retrieve the TAGE branch predictor's prediction confidence (High, Medium, Low). We then increment a confidence counter by 2 for a low, 1 for a medium, and 0 for a high confidence prediction. Whenever the counter exceeds a threshold, UDP assumes to be on the off-path, and no longer unconditionally emits prefetches. In particular, on the (assumed) off-path it will query UDP to determine whether a prefetch should be emitted. The confidence counter is reset on each branch recovery and BTB resteer. UDP also assumes to be on the off-path when the branch predictor predicts a branch as taken whose PC misses in the BTB.

When UDP assumes to be on the off-path, it only emits prefetches that have been learned to be useful. Therefore, it maintains a set of addresses that represent useful prefetch candidates. The set is populated whenever an on-path demand load hits an off-path prefetch (based on the confidence counter). Note that it is critical to only consider prefetches as useful that are hit by an on-path demand load as we do not want to learn prefetches that are consumed on the off-path only. To enable this, we add a *Seniority-FTQ* that holds off-path prefetch candidate blocks that are no longer in the FTQ, after having been consumed by frontend. The candidate is useful whenever the backend retires an instruction whose line address matches the prefetch candidate block. On a pipeline flush, blocks are removed from the Seniority-FTQ until the flush point. The Seniority-FTQ is much smaller than the reorder buffer (ROB). First, it holds coarse-grained fetch blocks instead of instructions, and second, it does not hold all blocks but only blocks containing a prefetch candidate.

As depicted in Figure 10, prefetch candidate blocks deemed useful by UDP are inserted into the *useful-set*. UDP queries the

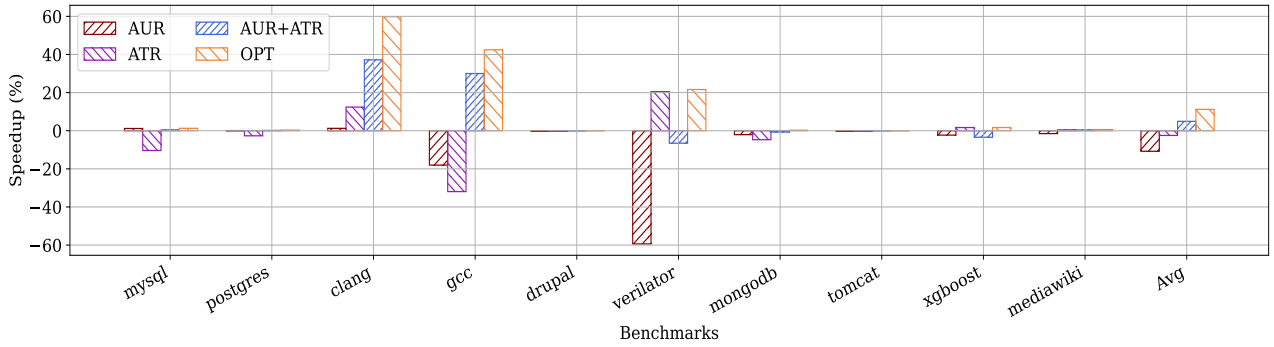

Fig. 11. IPC speedups of AUR, ATR, AUR+ATR, and the optimal: UFTQ-ATR-AUR achieves an average speedup of 4.9% (up to 37.2%) without significantly-negatively affecting an application.

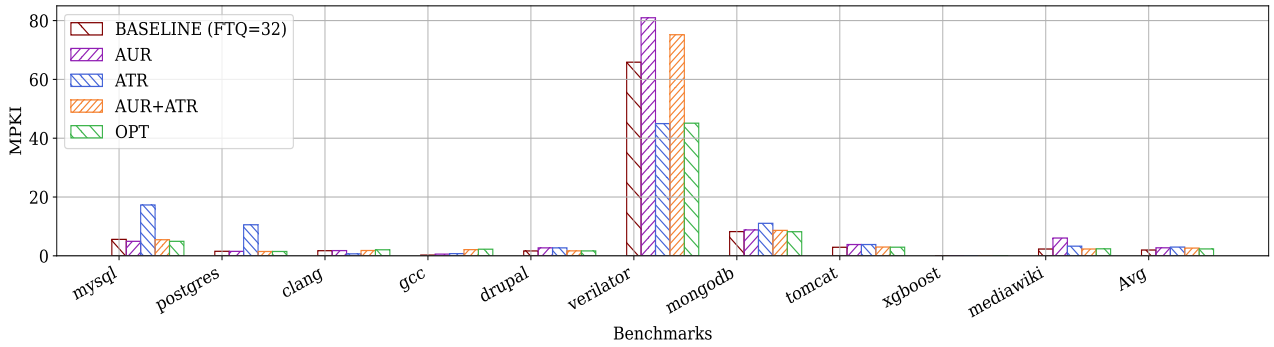

Fig. 12. Instruction cache misses per KI of AUR, ATR, AUR+ATR, and the optimal: UFTQ-ATR-AUR reduces 1.2% of icache misses on average and up to 28% for `verilator` with a larger FTQ where it emits more useful off-path prefetches.

useful-set whenever evaluating an off-path prefetch candidate. Only if the candidate is in the useful set and not in the icache, a prefetch is emitted. The useful-set can be implemented with an associative cache, however, as we only need to test whether an element is part of the set, we deploy space-efficient bloom filters [18]. While prefetch candidates refer to individual cache lines, we observe that frequently multiple consecutive lines are considered useful and thus inserted into the filter. We exploit this fact to improve the space efficiency of our technique further. Therefore, we introduce a small buffer that stores the last eight recent prefetch candidates before they get inserted into the filter. We then detect consecutive lines with monotonically increasing addresses and combine them into super-lines. We support combining two or four consecutive lines into a 2-block, respectively 4-block. As a result, a single block can represent either one, two, or four prefetch candidates, reducing the total number of candidates we need to store. We deploy three bloom filters holding one of the three block sizes. FDIP looks up a given prefetch candidate in all three filters, and if there is a hit, it will emit either one, two, or four prefetches. This optimization allows us to reduce the bloom filter size by $4\times$. In particular, we deploy a 1-block filter of 16k bits, a 2-block filter of 1k bits, and a 4-block filter of 1k bits. We also explored even longer consecutive sequences of prefetch candidates, however, such sequences are too infrequent to warrant another bloom filter. We configure the filters to have a false positive rate of 1% and utilize Open Bloom Filter [43] to generate optimal parameters (resulting in 6 hash functions). Our hardware implementation computes the six hash functions in parallel (1 cycle) and then reads the hash indexes from the banked Bloom Filter SRAM (1-6 cycles). The final step is to compare the six generated output bits with 0 to determine whether the address is part of the set. Efficient Bloom Filter hardware designs have been proposed by Sateesan [51]. Once one bloom filter is full and the unuseful ratio reaches 0.75 (we increment a counter of unuseful prefetches), we clear the filter. The total storage overhead of our design is 8KB.

V. Evaluation

We first describe our evaluation methodology and then present the performance results for the two proposed UFTQ and UDP techniques.

A. Methodology

We utilize Scarab, a cycle-accurate microarchitectural simulator, to evaluate our proposed techniques. As described in Section III, Scarab models a detailed superscalar out-of-order processor with a decoupled frontend, modern branch predictors, wrong-path simulation, and a detailed memory hierarchy. The detailed configuration parameters of the modeled system is provided in Table II. We evaluate 10 frontend-bound datacenter applications, utilizing a simpoint methodology. For each application, we simulate 10 (application-specific) 10M

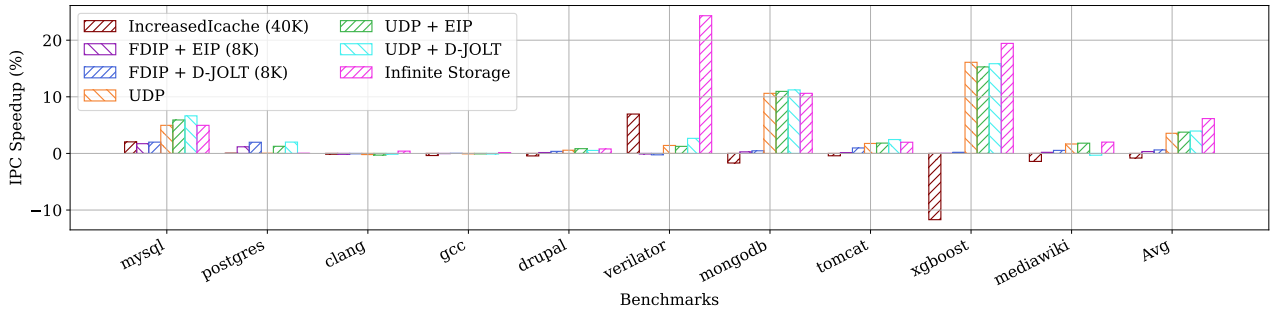

Fig. 13. IPC speedups of Infinite Storage and UDP: UDP with 8K bloom filter shows up to 16.1% IPC speedup and 3.6% on average.

instruction simpoints [26] which are aggregated according to their weight. Each simpoint is warmed up with 10M instructions. We evaluate UFTQ-ATR-AUR on 20% randomly chosen simpoints with the equation obtained by training 80% randomly chosen simpoints for memtraces and train half phases of PT traces and evaluate the other half instructions. We compare UDP to the top two instruction prefetchers from IPC-1 by porting the Champsim-based source codes to Scarab: Entangled Instruction Prefetcher (EIP) [49] and D-JOLT [42]. EIP is more optimized by eliminating wrong-path training on our simulation [48]. We compare UDP to a state-of-the-art FDIP baseline with different FTQ sizes (16,32,48,64) and different BTB sizes (4k,8k,16k).

B. UFTQ

Speedup. We now evaluate the IPC uplift provided by the proposed UFTQ-AUR, UFTQ-ATR, and UFTQ-ATR-AUR mechanisms optimizing the FTQ depth in an application-specific way, as described in Section III-B. We compare our three proposed techniques against the state-of-the-art baseline utilizing an FTQ with a fixed size of 32 and against an upper bound referred to as OPT. OPT represents an oracle approach that utilizes the optimal FTQ depth as determined in Section III. Figure 11 shows the achieved IPC speedup of three different mechanisms. As seen UFTQ-ATR-AUR provides a positive speedup of up to 37.2% for `clang` and `gcc` and an average speedup of 4.9%. It provides a positive speedup or negligible negative speedup for all applications and approaches the performance of OPT. In contrast, the UFTQ-AUR and UFTQ-ATR do not perform well. They either perform too aggressive or too conservative FTQ sizing decisions, as they cannot learn the three performance characteristics described in Section III. While Table III shows high correlation coefficients for the two techniques, they fail to efficiently adjust the FTQ size for `gcc`, and `verilator`. Note that the evaluated phases of the workload may have completely different performance characteristics. We focus on the model trained on workloads with completely different characteristics does not negatively impact other workloads.

For example, the evaluated phase of `gcc` and `mongodb` show negative speedups when it only follows UFTQ-AUR or UFTQ-ATR respectively as it emits too many harmful prefetches within 50% of off-path prefetches with the determined utility ratio of 0.65. On the other hand, `verilator` shows a negative speedup only on UFTQ-AUR because it misses useful off-path prefetches by preventing it from running ahead further due to the limited utility ratio. This incurs that UFTQ-AUR stops increasing the FTQ size too early at a smaller FTQ although `verilator` still has more than 60% of useful prefetches with a large FTQ. The combination of the two approaches, UFTQ-ATR-AUR, only prevents either inaccurate or untimely prefetches. In data center environments that operate millions of cores, even an IPC uplift of 4.9% provides substantial benefits in terms of operating expenditures and reducing carbon emissions.

Icache Miss Reduction and Improved Timeliness. The IPC speedup comes from a reduction of icache misses thanks to timely and accurate prefetches. We evaluate how well UFTQ-ATR-AUR prevents icache misses from being increased in Figure 12. At least for 20% of different application phases, UFTQ-ATR-AUR only shows less or similar icache misses to the baseline while UFTQ-ATR or UFTQ-AUR increases icache misses. It shows almost the same MPKI as OPT across all applications except for `verilator`. UFTQ-ATR-AUR increases icache misses for `verilator`, but it does not increase the misses when it only relies on UFTQ-AUR because it benefits from more timely fill-buffer hits. The reason why OPT still provides an additional IPC gain over UFTQ-ATR-AUR is also more timely prefetches.

C. UDP

Speedup. While UFTQ provides a substantial improvement at a low hardware cost, it cannot effectively utilize useful off-path prefetches. In particular, by restricting unuseful off-path prefetches through FTQ sizing, it also emits more potential of useful off-path prefetches timely. We now evaluate UDP, which can leverage useful off-path prefetches. Figure 13 shows the IPC uplift over the fixed 32-size FTQ baseline for UDP utilizing an 8K Bloom Filter. We also evaluate an upper-bound implementation. Infinite Storage leverages a useful set of infinite size which learns all useful off-path prefetches. All the UDP techniques utilize an FTQ of size 32. Within the same FTQ depth, it allows them to runahead further and prefetch more timely by only emitting prefetches if they are in the learned useful-set. As can be seen, UDP provides substantial performance gains of up to 16.1% for `xgboost` and 3.6% on

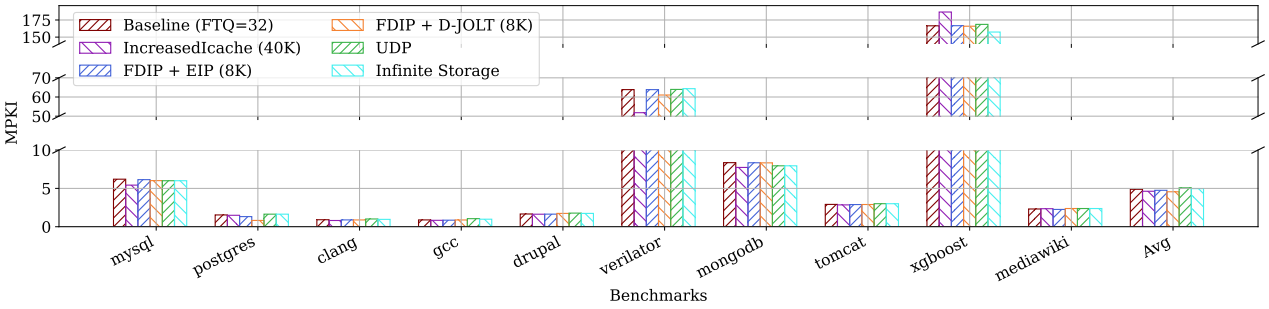

Fig. 14. Instruction cache misses per kilo instructions for Infinite Storage and UDP

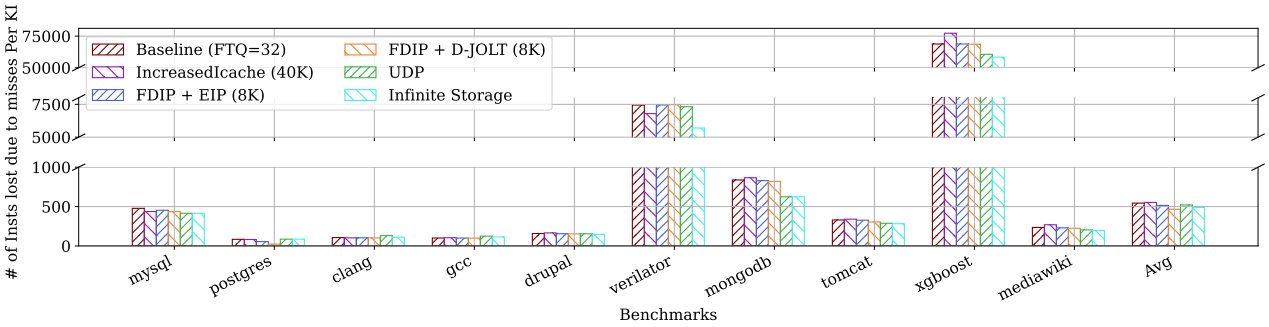

Fig. 15. Proportional to cycles spent due to icache misses per kilo instructions for Infinite Storage and UDP

average. The infinite storage technique provides even higher performance. For an ISO-storage comparison, we evaluate an increased icache (40K) and the Entangled Instruction Prefetcher (EIP - 8KB) which is one of top IPC-1 instruction prefetchers. Increasing the icache size rarely provides IPC gain, and EIP provides substantially lower IPC uplift for the same storage budget. The reason why EIP falls short compared to UDP is two-fold. First, EIP requires substantial meta-data. If provided with 100KB+ storage it performs well, however, 8KB are insufficient. Second, EIP is unaware of wrong-path execution and trains on all icache accesses utilizing additional storage to train for unuseful off-path prefetches.

Icache Miss reduction. Figure 14 shows the icache MPKI for the baseline and the five evaluated techniques. The MPKI does not substantially differ between the techniques, and for `xgboost` and `verilator`, the MPKI of UDP exceeds the baseline because it misses off-path useful prefetches. Even with almost the unchanged icache misses, the substantial IPC uplift is because of more timely prefetches. Figure 15 shows the number of instructions lost due to icache misses, proportional to the number of lost cycles due to misses. The usefulness technique reduces the number because it increases the timely and useful prefetches. UDP is a throttling technique that reduces the emitted prefetches. It works well to eliminate harmful unuseful prefetches, and eventually allows more useful prefetches to be loaded into icache in time. While it rarely reduces the number of icache misses, the performance benefit of UDP is obtained by substantially improving the timeliness of fill-buffer hits. As a result, UDP also improves power efficiency by reducing the number of emitted prefetches and off-chip memory traffic.

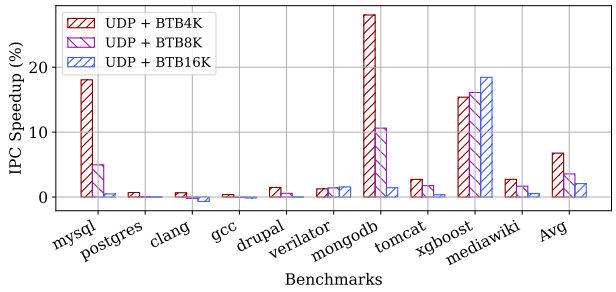

Fig. 16. UDP provides IPC uplifts on different BTB sizes.

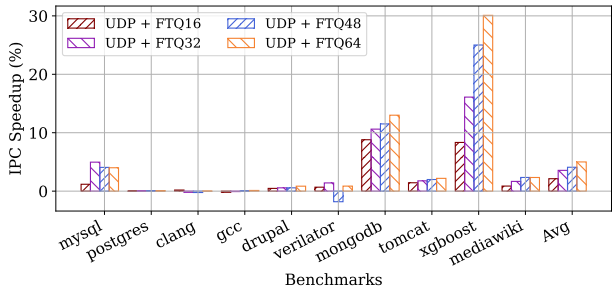

Fig. 17. UDP on top of different FTQ sizes (16, 32, 48, 64)

BTB/FTQ Sensitivity Analysis. Figure 16 shows UDP always provides IPC uplifts on different BTB sizes. UDP shows more speedups when the size of BTB is limited. Figure 17 shows that UDP also works on top of different FTQ sizes except for `verilator`. With a larger FTQ, the workload can further run ahead, leading to more aggressive prefetching where UDP fills and flushes the bloom filter fre-

quently. A conservative flushing policy with a higher threshold of unuseful ratio potentially works better for `verilator`-like workloads with plenty of useful off-path prefetches.

VI. Related Work

Google [16] and Facebook [56] have reported that large code footprint applications cause significant frontend stalls that intrude significant energy costs and carbon footprint emissions. Existing processor caches provide insufficient capacities to address this problem [15]. Reinman has introduced the decoupled frontend [46] and FDIP [47]. Together with Ishii's FDIP [28] these works represent the foundation and baseline for UDP.

Several prior works including Boomerang [38], Shotgun [37], Confluence [30], and Divide and Conquer Frontend Bottleneck [14] have introduced techniques to reduce BTB misses of large frontend applications to improve the efficacy of FDIP. While these techniques improve performance, they significantly increase the storage overheads of the BTB (Boomerang, Confluence) and the complexity of the frontend (Shotgun, D&C). In contrast to UDP, these techniques also do not address wrong-path prefetches induced by branch predictions, nor do they leverage potentially beneficial wrong-path prefetches. The entangled instruction prefetcher has been extended to reduce training of wrong-path instructions [48], we compare our work with the state-of-the-art instruction prefetchers. Finally, these techniques are orthogonal to UDP in that they can be combined to improve performance further.

A prior work [20] has also studied FTQ depth and aggressive decoupled-frontend. We show how our work complements this prior work by also considering the impact of wrong-path execution in the context of fetch-directed instruction prefetching. PDIP [25] associates a branch PC with a few high front-end critical (FEC) lines by leveraging techniques similar to EMISSARY [41] for identifying FEC lines. By selectively prefetching the lines with a high priority, PDIP effectively reduces FEC stalls with 43.5KB storage costs. In contrast, UDP further improves the prefetch timeliness of FEC lines by considering useful prefetches on the off-path while requiring a much lower storage cost of only 8KB.

Another line of work has focused on optimizing the storage efficiency of the BTB for reducing frontend resteers and wrong-path prefetches. Fagin [21] introduced partial tags in the BTB to improve storage efficiency. Kobayashi [36] introduces a 2-level BTB design. Perleberg [45], Lee [39], and PDede [55] reorganize the BTB for higher efficiency and introduce compression.

In addition to enhancing FDIP, prior works have also proposed stand-alone instruction prefetchers, including PIF [22], temporal instruction prefetching [23], D-Jolt [42], SHIFT [29], FNL+MMA [54], and the Entangling Instruction Prefetcher [49]. Ishii showed [28] that these stand-alone prefetchers provide no substantial benefit over FDIP. FDIP relies on highly accurate branch predictors to produce prefetching candidates, whereas stand-alone prefetchers have to predict these candidates utilizing a separate mechanism, that (1) introduces additional storage cost and hardware complexity and (2) does not increase the coverage of accurate prefetches substantially. UDP in contrast, leverages FDIP to produce prefetch candidates utilizing the efficacy of modern branch predictors.

Another line of work has proposed profile-guided software-hardware techniques to address frontend stalls. I-Spy [32] proposes software prefetch instructions to reduce instruction icache misses while Ripple [34] optimizes the replacement policy of the icache. Twig [31] introduces profile-guided BTB-prefetching to improve the performance of FDIP, while Whisper [33] improves branch predictor accuracy using software hints. All these techniques show significant benefits but also suffer from shortcomings. First, they require ISA changes which are difficult to justify for performance improvements. UDP, on the other hand, is a pure hardware technique that is ISA agnostic and transparent. Secondly, these prior works require profiling and re-compilation of all binaries. A sophisticated software environment needs to be maintained that only a few data center operators can justify at this point.

VII. Conclusion

This paper analyses the efficacy of fetch-directed instruction prefetching (FDIP), which is utilized in virtually all modern architectures with a decoupled frontend, including most processors by AMD, Intel, and ARM. We determined that state-of-the-art implementations of FDIP fall significantly short of optimal performance and then determine the root causes for said inefficiency. In particular, we quantify the effect of untimely prefetches and inaccurate off-path prefetches, observing that applications exhibit fundamentally different characteristics rendering one-size-fits-all solutions inefficient. Furthermore, we measure the usefulness of off-path prefetches after branch mispredictions and leverage these results to design mechanisms improving the performance of FDIP. In particular, we introduce UFTQ, a technique to dynamically adapt the prefetch aggressiveness of FDIP by tuning the size of the fetch target queue. We then further improve performance by introducing UDP, a technique that learns the usefulness of prefetch candidates to enable even more aggressive prefetching (improving timeliness) while eliminating virtually all harmful inaccurate prefetches. Our hardware mechanisms introduce moderate hardware overheads and implementation complexity while improving IPC performance by up to 16.1% and by 3.6% on average.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and helpful feedback. This work was generously supported by Intel's Center for Transformative Server Architectures (TSA), the PRISM Research Center, a JUMP Center cosponsored by SRC and DARPA, and NSF grant #2111688. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

APPENDIX

A. Abstract

This artifact contains all the source code for the UDP in Scarab simulator on the x86 sunnycove-like architecture and its tools to run datacenter applications with Scarab. These tools allow users to launch Scarab simulations and reproduce the main results. We open-source the extended Scarab and the tool that creates a docker container where all the benchmarks are ready to be simulated and plots the results. Users can reproduce the results in Figures 13, 14, 15, 16, and 17.

B. Artifact check-list (meta-information)

- **Program:** Scarab simulator and Docker tool
- **Data set:** All the traces are available in the docker container.
- **Run-time environment:** Ubuntu 20.04.5
- **Hardware:** Intel x86_64 processor.
- **Execution:** Automated by tooling.
- **Output:** Graphs of IPC speedups, Icache MPKI, # of instruction lost due to Icache misses of UDP (Fig 13-17).
- **Experiments: Automated by tooling.**
- **How much disk space required (approximately)?:** 50GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes to install Docker and build a docker image
- **How much time is needed to complete experiments (approximately)?:** 24 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT license
- **Data licenses (if publicly available)?:** MIT license
- **Workflow framework used?:** Yes
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.11068829

C. Description

1) How to access: All source code is open-source and available on GitHub. The workflow requires cloning https://github.com/Litz-Lab/Scarab-infra.git which is a tool to automate the workflow of installing Scarab, all the dependent libraries, benchmark traces and plotting the figures in the paper.

2) Hardware dependencies:

- **Intel x86_64 processor.** Scarab leverages Intel's PIN, a dynamic binary instrumentation framework for the IA-32, x86-64, and MIC instruction-set architectures.
- **> 20 multi-core processor preferred.** Scarab simulations are launched in parallel for all the simpoints for a given single configuration. The number of simpoints of the workloads for this artifact is from 2 to 36, depending on the workload. In theory, at most, 36 processes can be executed in parallel.
- **> 50GB storage space.** The size of the docker image (23G), including all the benchmark traces, and the space for simulation results will reach 50G.
- **> 50GB RAM.** Parallel simulation at some point loads 50GB traces.

3) Software dependencies: Docker, Python3, Python library versions in https://github.com/Litz-Lab/Scarab-infra/blob/ISCA2024-UDP/isca2024_udp/plot/requirements.txt

4) Data sets: All the traces and simpoints are available inside the docker container at `/simpoint_traces`.

D. Installation

1) Install the Docker engine if not installed on your local machine by reviewing https://docs.docker.com/engine/install/.
2) Clone https://github.com/Litz-Lab/Scarab-infra.git and checkout `ISCA2024-UDP` branch:

```
git clone -b ISCA2024-UDP https://github.com/Litz-Lab/Scarab-infra.git
```

3) Step-by-step procedures have been explained in the https://github.com/Litz-Lab/Scarab-infra/blob/ISCA2024-UDP/README.ISCA.md on ISCA2024-UDP branch. Step 1 is the installation part.

E. Experiment workflow

Step 2 in `README.ISCA.md`. We provide an automated workflow to validate the main results in the paper from scratch. The user launches Scarab simulations for all the configurations and workloads described in `./isca2024_udp/isca.json` and `./isca2024_udp/isca.pt.json`.

F. Evaluation and expected results

Step 3 in `README.ISCA.md`. Running `plot_figure.sh` generates the UDP plots (Fig13-17). The figures are generated at the same path as `plot_figures.sh` (Figure13.pdf, Figure14.pdf, Figure15.pdf, Figure16.pdf, and Figure17.pdf).

G. Experiment customization

The workflow can be even more parallelized if more HW resources (e.g. # of cores, RAM) are available. The divided experiment file descriptors `isca_cust1.json` and `isca_cust2.json` are available in the same path of `isca.json`. The way to manually parallelize Step 2 is the following.

```
./run.sh -o /home/$USER/isca2024_home -s 4 -e isca_cust1
./run.sh -o /home/$USER/isca2024_home -s 4 -e isca_cust2
./run.sh -o /home/$USER/isca2024_home -s 5 -e isca.pt
```

H. Notes

For more information about Scarab v2.0 release, please visit the GitHub page: https://github.com/Litz-Lab/scarab

REFERENCES

[1] "Apache tomcat," https://tomcat.apache.org/.

[2] "Clang c language family frontend for llvm," [Online; accessed 19-Nov-2021]. [Online]. Available: https://clang.llvm.org/

[3] "Github - performance tools for mongodb," [Online; accessed 23-Feb-2024]. [Online]. Available: https://github.com/mongodb/mongo-perf

[4] "Github - scriptable database and system performance benchmark," [Online; accessed 21-Nov-2023]. [Online]. Available: https://github.com/akopytov/sysbench

[5] "Higgs dataset at mlpysics portal," https://public.dhe.ibm.com/software/mktsupport/techdocs/power4.pdf.

[6] "Ibm power4 system microarchitecture," https://public.dhe.ibm.com/software/mktsupport/techdocs/power4.pdf.

[7] "Mysql," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.mysql.com

[8] "Postgresql: The world's most advanced open source database," [Online; accessed 19-Nov-2021]. [Online]. Available: https://www.postgresql.org/

[9] "Scarab," https://github.com/hpsresearchgroup/scarab.

[10] "Scarab," https://github.com/Litz-Lab/scarab.

[11] "Scarab-infra," https://github.com/Litz-Lab/Scarab-infra.

[12] "Spec cpu 2017 standard performance evaluation corporation," [Online; accessed 21-Nov-2023]. [Online]. Available: https://www.spec.org/cpu2017/

[13] "Verilator," https://www.veripool.org/wiki/verilator.

[14] A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Divide and conquer frontend bottleneck," in *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

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

[16] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th ISCA*, 2019.

[17] K. Banker, D. Garrett, P. Bakkum, and S. Verch, *MongoDB in action: covers MongoDB version 3.0*. Simon and Schuster, 2016.

[18] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *European Symposium on Algorithms*. Springer, 2006, pp. 684–695.

[19] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization*, 2003.

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

[21] B. Fagin, "Partial resolution in branch target buffers," *IEEE Transactions on Computers*, vol. 46, no. 10, pp. 1142–1145, 1997.

[22] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *International Symposium on Microarchitecture*, 2011.

[23] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *International Symposium on Microarchitecture*, 2008.

[24] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, 2017.

[25] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Seznec, D. Tullsen, and D. 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*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 846–861. [Online]. Available: https://doi.org/10.1145/3620665.3640394

[26] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[27] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the 18th Annual International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 2004, p. 1–11. [Online]. Available: https://doi.org/10.1145/1006209.1006211

[28] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching: An industry perspective," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2021.

[29] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," 2013.

[30] ——, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 166–177.

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

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

[33] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 19–34.

[34] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-guided instruction cache replacement for data center applications," in *Proceedings (to appear) of the 48th International Symposium on Computer Architecture (ISCA)*, ser. ISCA 2021, Jun. 2021.

[35] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer Architecture Letters*, vol. 15, no. 1, pp. 45–49, 2016.

[36] R. Kobayashi, Y. Yamada, H. Ando, and T. Shimada, "A cost-effective branch target buffer with a two-level table organization," in *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*, 1999.

[37] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 30–42, 2018.

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

[39] Lee and Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, 1984.

[40] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[41] N. P. Nagendra, B. R. Godala, I. Chaturvedi, A. Patel, S. Kanev, T. Moseley, J. Stark, G. A. Pokam, S. Campanoni, and D. I. August, "Emissary: Enhanced miss awareness replacement policy for l2 instruction caching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589097

[42] T. Nakamura, T. Koizumi, Y. Degawa, H. Irie, S. Sakai, and R. Shioya, "D-jolt: Distant jolt prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, 2020.

[43] A. Partow, "Open bloom filter," "http://www.partow.net/programming/hashfunctions/index.html.

[44] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala *et al.*, "The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.

[45] C. H. Perleberg and A. J. Smith, "Branch target buffer design and optimization," *IEEE transactions on computers*, vol. 42, no. 4, pp. 396–412, 1993.

[46] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," *ACM SIGARCH Computer Architecture News*, vol. 27, no. 2, pp. 234–245, 1999.

[47] 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*. IEEE, 1999, pp. 16–27.

[48] A. Ros and A. Jimborean, "Wrong-path-aware entangling instruction prefetcher," *IEEE Transactions on Computers*, vol. 73, no. 02, pp. 548–559, feb 2024.

[49] ——, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, 2020.
[50] J. Rupley, "Samsung exynos m3 processor," *IEEE Hot Chips*, vol. 30, 2018.
[51] A. Sateesan, J. Vliegen, J. Daemen, and N. Mentens, "Hardware-oriented optimization of bloom filter algorithms and architectures for ultra-high-speed lookups in network applications," *Microprocessors and Microsystems*, vol. 93, p. 104619, 2022.
[52] A. Seznec, "Tage-sc-l branch predictors," in *JILP-Championship Branch Prediction*, 2014.
[53] ——, "Exploring branch predictability limits with the mtage+ sc predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016, p. 4.
[54] ——, "The fnl+mma instruction cache prefetcher," in *IPC-1 - First Instruction Prefetching Championship*, 2020.
[55] N. K. Soundararajan, P. Braun, T. A. Khan, B. Kasikci, H. Litz, and S. Subramoney, "Pdede: Partitioned, deduplicated, delta branch target buffer," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 779–791.
[56] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "Softsku: Optimizing server architectures for microservice diversity@ scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 513–526.
[57] D. Suggs, M. Subramony, and D. Bouvier, "The amd "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. bo45–52, 2020.
[58] Wikipedia contributors, "Drupal — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Drupal&oldid=989582664, 2020, [Online; accessed 23-November-2020].
[59] ——, "Mediawiki — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=MediaWiki&oldid=989993176, 2020, [Online; accessed 23-November-2020].
[60] ——, "Xgboost — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/XGBoost, 2023, [Online; accessed 26-April-2023].