Alternate Path μ-op Cache Prefetching

Sawan Singh*, Arthur Perais†, Alexandra Jimborean*, Alberto Ros*

*Computer Engineering Department, University of Murcia, Murcia, Spain
†Univ. Grenoble Alpes, CNRS, Grenoble INP, TIMA, Grenoble, France
Email: singh.sawan@um.es, arthur.perais@univ-grenoble-alpes.fr, alexandra.jimborean@um.es, aros@ditec.um.es

***Abstract*—Datacenter applications are well-known for their large code footprints. This has caused frontend design to evolve by implementing decoupled fetching and large prediction structures – branch predictors, Branch Target Buffers (BTBs) – to mitigate the stagnating size of the instruction cache by prefetching instructions well in advance. In addition, many designs feature a micro operation (μ-op) cache, which primarily provides power savings by bypassing the instruction cache and decoders once warmed up. However, this μ-op cache often has lower reach than the instruction cache, and it is not filled up speculatively using the decoupled fetcher. As a result, the μ-op cache is often over-subscribed by datacenter applications, up to the point of becoming a burden.**

This paper first shows that because of this pressure, blindly prefetching into the μ-op cache using state-of-the-art standalone prefetchers would not provide significant gains. As a consequence, this paper proposes to prefetch only critical μ-ops into the μ-op cache, by focusing on execution points where the μ-op cache provides the most gains: Pipeline refills. Concretely, we use hard-to-predict conditional branches as indicators that a pipeline refill is likely to happen in the near future, and prefetch into the μ-op cache the μ-ops that belong to the path opposed to the predicted path, which we call *alternate* path. Identifying hard-to-predict branches requires no additional state if the branch predictor confidence is used to classify branches. Including extra *alternate* branch predictors with limited budget (8.95KB to 12.95KB), our proposal provides average speedups of 1.9% to 2% and as high as 12% on a subset of CVP-1 traces.

I. Introduction

The number of processor stall cycles attributed to the frontend in datacenter workloads is reported to reach 23.5% [13], which is significant as one would rather expect stall cycles to stem mostly from (i) waiting on data in the backend and (ii) branch mispredictions. As a result, both industry and academia have proposed several solutions to mitigate these stalls, including instruction prefetching [16], [47], [47], [55], [56], [59], [70], improving branch predictors [34]–[36], [44], [65], [68], [71] and using larger branch target buffers (BTBs) [19], [25], [32], [64].

Despite continuous advancements, increasing pressure is applied to these structures by growing code footprints, especially in datacenter-class workloads. These workloads run deep stacks and their code footprint can exceed current L1 instruction cache (L1I) capacities by two orders of magnitude [13]. Furthermore, it is predicted that their code footprint will keep increasing at the rate of 20% per year [13]. Not only does the code not fit in the L1I cache, but the large BTBs also struggle to provide enough reach to track all branches [12]–[14], [21], [28], [29], [38], [39]. On one hand, L1I misses contribute to performance degradation by stalling the frontend while an instruction is being retrieved from the memory system. This is mitigated by Decoupled Fetching (or Fetch Directed Prefetching, FDP) [55], [56], in which fetch address generation and instruction retrieval from the memory system are decoupled. This allows fetch address generation to run ahead during L1I misses, enabling the overlap of instruction misses and performing instruction prefetching based on branch direction and target predictions. On the other hand, FDP relies on the BTB to guide instruction fetch, that is, the burden of caching information about the whole code footprint is shifted from the L1I to the BTB, which, despite steady growth across commercial processor generations, often struggles to capture the whole code footprint. BTB misses cause potentially wrong path instructions to be fetched from the L1I and inserted in the pipeline, causing additional pipeline re-steers once the taken branches are identified in decode.

Last but not least, large code footprints exceed the microarchitectural operation (μ-op) cache capacity, limiting its usefulness. A μ-op cache is currently implemented in many processor designs used in datacenters [8], [61]. This structure caches decoded instructions (μ-ops) instead of architectural instructions and serves two purposes. The first is power efficiency, as consistently hitting in the μ-op cache avoids accessing the L1I and bypasses the decoders. The second is performance, as the throughput of the μ-op cache is generally higher than the one of the "slow path" decoders. This, combined with a shortened pipeline length when hitting in the μ-op cache, can reduce the average cost of branch mispredictions. However, modern μ-op caches generally have smaller reach than instruction caches. For instance, Amd Zen4 can cache up to 6.75Kops [8], amounting to 24.9KB worth of x86 instructions (if (1) all instructions are assumed to decode to a single μ-op and (2) one x86 instruction occupies 3.7B on average, as in SPEC CPU 2k6 INT [15]). As a result, in the context of large instruction footprint workloads, the μ-op cache struggles to retain enough useful μ-ops to provide the power and performance improvements it was designed for. Even worse, switching from the μ-op cache as a source of μ-ops to the decoders can incur a penalty [57], even on recent microarchitectures such as Amd Zen [8]. In other words, continuously alternating between hits and misses can actually degrade performance.

979-8-3503-2658-1/24/$31.00 ©2024 IEEE
DOI 10.1109/ISCA59077.2024.00092

In this paper we argue that the *µ-op cache is too small for datacenter applications and cannot deliver the power savings and additional throughput it has been designed for*. Nevertheless, increasing its size to address large code footprints is not more feasible than increasing the size of the L1I cache, and most datacenter commercial processors remain limited to a 32KB L1I [8], [61]. Rather, we argue that the µ-op cache insertion policy should take into account the over-subscription level of the µ-op cache, such that some of the benefits are retained even for large-footprint workloads.

This paper first quantifies the inability of the µ-op cache to accommodate the ever-increasing code footprint of datacenter workloads. We then emphasize the criticality of the instructions fetched from the not predicted (alternate) path immediately after a branch misprediction and show that a significant fraction of the µ-op cache benefits can be retained in large code footprint workloads by maximizing µ-op cache hits on pipeline refills. Finally, we introduce UCP (µ-op cache prefetching), a microarchitecture that *selectively prefetches alternate path instructions* into the µ-op cache. Alternate-path prefetching is triggered when hard-to-predict (H2P) conditional branches are fetched, thus accelerating pipeline refill should the branch mispredict. This contrasts with prior work where alternate path instructions must reach the execution stage to become useful, thus consuming significant resources, only to be discarded if they are in fact not needed [11], [17].

UCP provides benefit from the alternate path by prefetching alternate-path µ-ops in the µ-op cache and leveraging the prefetched instructions to speedup pipeline (re)fills. By prefetching selectively, the prefetched µ-ops remain in the µ-op cache longer, and can be fetched not only for the current instance of an H2P branch, but also for upcoming executions. In other words, even if the current instance of the H2P branch is not mispredicted, by the H2P definition, it is likely to be mispredicted in a subsequent execution. Hence, caching the alternate path is highly likely to be useful in the near future. This work makes the following contributions:

- We quantify the µ-op cache hit rate and impact on performance for applications with large code footprints.
- We propose selective µ-op cache prefetching by caching instructions from the alternate path (i.e. the opposite of the predicted path) for H2P branches.
- We improve on state-of-the art branch prediction confidence estimation by building on storage-free confidence estimation using TAGE prediction counters [67].
- We show that for prefetching the alternate path, a low-storage alternate conditional branch predictor suffices.
- Our results show that µ-op cache prefetching can achieve 2% IPC (resp. 1.9%) IPC improvement (geomean) with modest hardware overhead 12.95KB (resp. 8.95KB), and increases the proportion of workloads that benefit from the µ-op cache to 90%, from 80.7%.

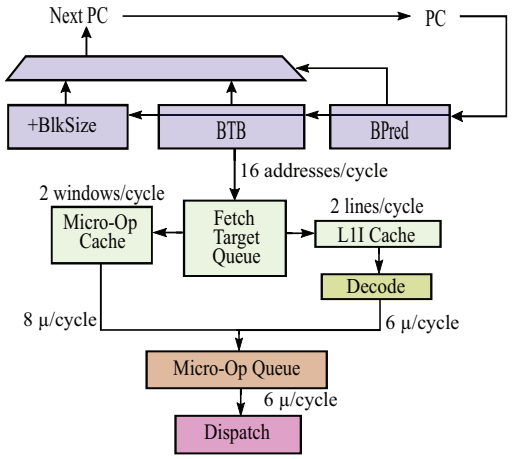


Fig. 1: Processor frontend

II. Background

This section offers an overview of the frontend of a modern processor featuring a µ-op cache, illustrated in Fig. 1. The addresses of the instructions to fetch are generated by the branch prediction unit (BPU). The BPU consists of a branch direction predictor, a BTB, an indirect target predictor, and a return address stack (RAS). Up to 16 instruction addresses can be generated by the BPU per cycle [5], which are placed in the fetch target queue (FTQ). Addresses in the FTQ are used to index either or both the L1I cache and the µ-op cache, depending on the current frontend operating mode. The L1I cache contains recently-used encoded architectural instructions, while the µ-op cache holds µ-ops recently generated by decoding instructions fetched from L1I.

The frontend is able to operate in two modes [73]. In *stream* mode, the FTQ only queries the µ-op cache. On a hit, µ-ops are directly sent to the µ-op queue. This represents the fast path and saves power as the L1I and decoders are bypassed. All the entries from the µ-op queue move to the dispatch queue to be allocated and issued in the processor backend. On a µ-op cache miss, the mode is switched to *build* mode. The L1I is then queried to provide instructions that will flow through the decoders to generate µ-ops, before being inserted in the µ-op queue. As instructions are decoded, a hardware block is responsible for building µ-op cache entries following specific rules that dictate the termination of a µ-op cache entry [43]: (1) A predicted taken branch (2) Crossing an L1I line boundary (3) Exceeding a statically defined number of (a) µ-ops (b) immediate or displacement fields (c) micro-coded µ-ops.

In *build* mode, the frontend keeps querying both the L1I and the µ-op cache in parallel until encountering a number of consecutive hits in the µ-op cache, upon which the frontend switches back to *stream* mode to save power. L1I hits therefore represent the slow path, as architectural instructions need to be decoded. Furthermore, continuously alternating between the two modes introduces latency overhead [3], [57].

The µ-op cache has been primarily designed for power savings [73], by holding the µ-ops of frequently executed

instructions. However, in modern x86 processors, its role goes beyond that. Indeed, since decoding multiple x86 instructions in parallel is a hard problem, decode width remains limited to 4-5 architectural instructions even in aggressive designs. However, the µ-op cache width can exceed this limit at minimal cost, by caching more µ-ops per entry. For instance, AMD Zen4 can provide up to 9 *macro ops*[1] per cycle from the µ-op cache, while it is limited to decoding 4 architectural instructions per cycle, which generally yield fewer than 9 *macro ops* [8]. Therefore, from a performance standpoint, the larger width combined with the shortened frontend length stemming from bypassing decoders makes the µ-op cache an efficient pipeline (re)fill accelerator, as long as the requested µ-ops are found in the µ-op cache. We emphasize that caching µ-ops is not limited to microarchitectures implementing complex instruction sets. For instance, the ARM Neoverse V2 microarchitecture features a 1.5K-entry decoded cache [30].

III. Motivation

This section presents a study of the performance impact of the µ-op cache for –mostly datacenter– applications featuring a large code footprint. In an attempt to justify why performance remains far from ideal, we analyze two complementary metrics: (1) the µ-op cache hit rate and (2) the number of switches between the *build* and *stream* modes per kilo instructions (PKI). Next, we show that simply increasing the µ-op cache size does not translate to proportional performance gains. To this end, we conduct a sensitivity study with respect to the µ-op cache size, its impact on the hit rate and performance. We then analyze whether state-of-the-art L1I instruction prefetching techniques, extended to prefetch also in the µ-op cache, can sufficiently increase the µ-op cache hit rate to close the performance gap with an ideal µ-op cache. Finally, we demonstrate that targeting certain *critical* instructions and prefetching those in the µ-op cache can noticeably improve performance despite modestly improving the µ-op hit rate. This is enough to prevent the µ-op cache from degrading performance in datacenter applications with large instruction footprints.

A. The impact of the µ-op cache on performance

We use ChampSim to model an Alder Lake pipeline and we analyze the behavior of datacenter applications using the "secret" set of 1^{st} Championship Value Prediction (CVP-1) [50] traces (model and trace details can be found in Section V). In the traces, 90% (resp. 100%) of the most frequently fetched 64B cache lines represent 120KB (resp. 720KB) of static code on average, for only 100M instructions. This highlights that pressure on the L1I and µ-op cache is significant. It should be noted that the CVP-1 traces are ARMv8 traces. Thus, instructions fit and are aligned on 4 bytes. For simplicity, this work assumes that one ARMv8 instruction translates to a single µ-op and implement 8 µ-op in each µ-op cache entry, with an entry covering 32B.

[1]Amd translates x86 instructions to one or more *macro ops*. *Macro ops* are therefore decoded instructions.

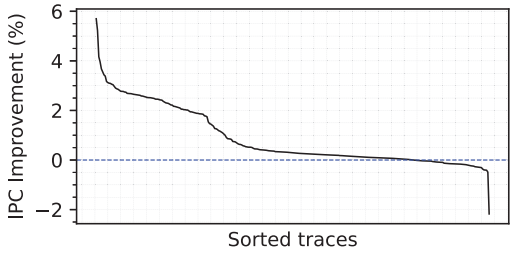


Fig. 2: IPC Improvement of a 4Kops µ-op cache normalized with a no µ-op cache baseline for CVP-1 traces

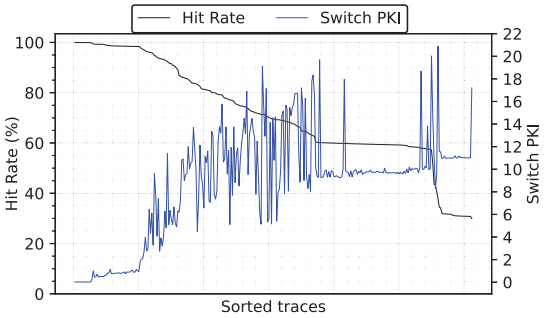


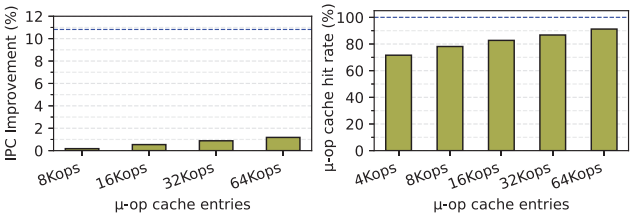
Fig. 3: µ-op cache hit rate and switch PKI across the CVP-1 traces. Sorted by hit rate.

In practice, choosing how many µ-op should reside in an entry depends on the actual µ-op set implemented by the microarchitecture as well as the average (or wost case) number of µ-op per architectural instruction. However, to the best or our knowledge, neither pieces of information are publicly available for any state-of-the-art microarchitecture. Moreover, we assume enough immediate/displacement storage for two branch targets per entry, but optimistically do not implement a limit on other immediates as the traces do not contain the information. The same applies for micro-coded µ-ops. If more than two branches are required, a new entry that covers the same 32B region is started and will be inserted in another way of the same set [57].

Fig. 2 shows the instructions per cycle (IPC) improvements when using a 4Kops µ-op cache over not using a µ-op cache. While beneficial for 80.7% of the traces, the µ-op cache degrades performance in 19.3% of the traces. The slowdown comes from the mode switching penalty when alternating between µ-op cache hits and misses, which confirms that the µ-op cache is only beneficial for applications that exhibit long enough streams of consecutive hits [3], [5].

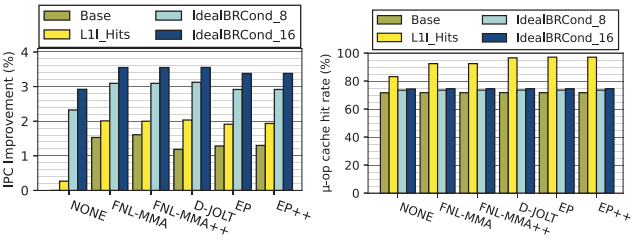
In fact, the µ-op cache is often unable to accommodate the code footprint, as illustrated in Fig. 3 which shows the per-instruction hit rate of the CVP-1 traces in a 4Kops µ-op cache.

The average (amean) µ-op cache hit rate reported in Fig. 3 is 71.6%, with very few applications reaching 99%. In the worst cases, the hit rate is as low as 30.7%. Overall, we found that about half of the applications considered in this work exhibit a hit rate of 70% or less, suggesting that the code footprint of datacenter workloads overwhelms the µ-op cache. Additionally, applications showing less than 95%



(a) Speedup (b) µ-op cache hit rate

Fig. 4: Analysis increasing the µ-op cache size. The blue line represents an ideal µ-op cache.



(a) Speedup (b) µ-op cache hit rate

Fig. 5: Instruction prefetchers versus alternate path

hit rate suffer from significantly more mode switches, thus partially offsetting the benefits of using a µ-op cache. In fact, 19.3% of the traces slightly lose performance when a µ-op cache is implemented.

B. Increasing the µ-op cache size

As a second step, we analyze whether larger µ-op caches would sufficiently increase the hit rate and therefore performance. Fig. 4 reports IPC and µ-op cache hit rate when increasing the µ-op cache size from 4Kops to 64Kops. Doubling the size from 4Kops to 8Kops increases the hit rate from 71.6% to 78.2% and yields an IPC improvement of only 0.18%, with a maximum improvement of 1.3% and a maximum slowdown of -3.6%. Even a 16x larger µ-op cache provides IPC improvements of only 1.2% with a hit rate of 91.2%. This is still far from the average performance gain of an ideal µ-op cache, which stands at 10.8% (blue line).

We conclude that merely increasing the µ-op cache size is insufficient to reach significant performance gains. While the theoretical limit is as high as 36%, in practice, growing the µ-op cache would increase its latency and power consumption. As a result, the next tool in the microarchitect toolbox is to keep the µ-op cache small, but to prefetch µ-ops, either through a dedicated prefetcher or through FDP.

C. State-of-the-art L1I prefetchers versus alternate path

As a first approximation, one would assume that instruction prefetching through decoupling branch prediction and fetch (FDP) would be sufficient to hide instruction misses, should branch prediction be able to run ahead far enough. However, FDP can only prefetch *predicted*-path instructions: On a branch misprediction, long latency instruction fetches can harm performance since the correct path was not prefetched. Conversely, standalone L1I prefetchers are able to issue prefetches for *alternate* path instructions even though the pipeline is on the *predicted* path. This is a fundamental advantage and partially explains why standalone prefetchers can bring additional gains on top of FDP [32]. We compare the impact of employing state-of-the-art L1I prefetchers to (ideally) prefetch in the µ-op cache and the potential benefits of prefetching critical instructions, i.e. the alternate path (see Fig. 5). Note that contrary to L1I prefetching, µ-op cache prefetching will require either sharing decoders with the decode stage, or implementing dedicated decoders. In this experiment, we assume dedicated decoders.

We evaluate three leading L1I prefetchers from the 1st Instruction Prefetching Championship (IPC1) [2]: FNL-MMA [70] (including its latest available version, labeled as FNL-MMA++), D-JOLT [47], and Entangling Prefetcher (EP) [58] (both its cost-effective version [59] and its further optimized version [60], labeled as EP and EP++, respectively). Fig. 5a presents the improvements in IPC with no L1I prefetcher as a baseline, whereas Fig. 5b illustrates the µ-op cache hit rate. The figures report numbers for three configurations for each L1I prefetcher:

a) Standalone L1I Prefetcher (Base): This configuration confirms that adding a standalone L1I prefetcher modestly improves performance (between 1.1% and 1.6%) over the No Standalone L1I Prefetcher (first bar, first group). Since the L1I prefetchers only target the L1I, the µ-op cache hit rate remains unchanged across the Base configurations. The No Standalone L1I Prefetcher serves as a baseline for all other configurations presented in this figure.

b) All L1I Hits are µ-op Cache Hits (L1I-Hits): This configuration is akin to immediately inserting all cache lines obtained through decoupled fetching in the µ-op cache. It achieves a µ-op cache hit rate as high as 97% when using the EP L1I prefetcher on top of FDP, while the IPC gain increases from 1.3% to 1.9%.

c) All Instructions after a Conditional Branch Misprediction are µ-op Cache Hits (IdealBRCond-8/16): Building on the insight that prefetching on the *alternate* path can provide benefits, we study *IdealBRCond-8*, where all instructions after a conditional branch misprediction are marked as µ-op cache hits, until 8 conditional branches have been fetched. *IdealBRCond-16* is similar to *IdealBRCond-8* but marks all instructions as µ-op cache hits until 16 conditional branches have been fetched. This is akin to perfectly prefetching µ-ops after branch mispredictions. *IdealBRCond-8* provides a better opportunity for improvement than *L1I-Hits* at 2.3%, despite the hit rate increase being quite modest (from 71.6% in the baseline to 73.5%). When up to 16 branches are considered, the IPC increase is 2.9% (min 0% & max 13.6%). This improvement comes from expediting instruction dispatch after a branch misprediction, that is, on a pipeline refill.

The previous experiments highlight that an efficient approach to µ-op cache prefetching would be to focus on alternate path instructions that follow a branch misprediction.

Indeed, on one hand, larger µ-op sizes and ideal prefetching bring only moderate speedups, despite significant hit rate increases: *L1I-Hits* achieved a minimum hit rate of 83.2% without any L1I prefetcher and a maximum of 97% when using EP as the L1I prefetcher. Yet, the pipeline often cannot consume µ-ops at the rate at which the µ-op cache can provide them, limiting the usefulness of a higher hit rate from the performance point of view. Furthermore, maximizing µ-op cache hit rate can still struggle to push performance up if the streams of hits are too short and the mode switching penalty is paid often. On the other hand, focusing on pipeline refills has higher potential for performance gains, as during a refill, the backend is starved for instructions more often. Moreover, ensuring that consecutive basic blocks after a pipeline flush are in the µ-op cache provides smooth µ-op delivery without triggering any switch to the L1I & decoder pipeline. Finally, prefetching only a small portion of the code decreases the likelihood of thrashing the µ-op cache.

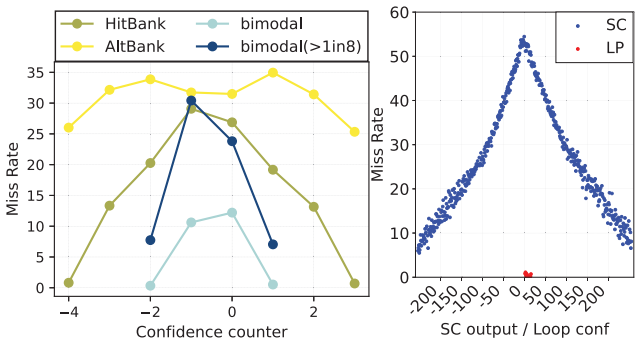
IV. Alternate Path µ-op Cache Prefetching

Motivated by our previous study on traditional prefetching versus alternate path prefetching, we propose to trigger alternate path µ-op cache prefetching (UCP) on low-confidence conditional branch predictions. By targeting conditional branches, there is a unique alternate path to follow, which simplifies our detection and prefetching mechanisms. The first step to enable UCP consists in detecting low-confidence conditional branch predictions, which we treat as hard-to-predict (H2P) branches. Second, we start generating the alternate path addresses, prefetch their corresponding instructions, decode, and store them in the µ-op cache, without hindering the progress of the predicted path. Finally, the last step consists in determining when the alternate path is unlikely to be useful, in order to stop prefetching and prevent µ-op cache pollution.

A. Branch Prediction Confidence

This section presents the mechanism to estimate the confidence of conditional branches, which we use both to initiate and stop alternate path prefetching.

Our predictor is based on the confidence estimation heuristic that can be built within the TAGE branch predictor [67], which determines the confidence of a direction prediction (low, medium, and high) based on the table that provided the prediction and the value of the saturating counter. TAGE employs 37 [68] tagged tables and a Bimodal base predictor. A prediction is provided either by Bimodal, or by one of the tagged tables, but TAGE distinguishes between the HitBank and the AltBank. Both are tagged tables that match the context, but HitBank is the one using the longest global branch history, while AltBank is the one using the second longer global branch history. In the initial heuristic, high confidence predictions are the ones that find the counter saturated regardless of which table provides it, unless the prediction comes from the bimodal table and there was at least one misprediction in the last eight predictions provided by the bimodal table.



(a) Bimodal and tagged tables (b) SC and LP

Fig. 6: Average misprediction rate for different components in a 64KB TAGE-SC-L, per output value

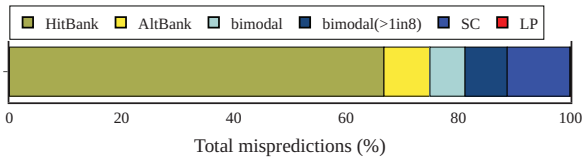


Fig. 7: Contribution of 64KB TAGE-SC-L components to mispredictions

Fig. 6 displays the average miss rate of a state-of-the-art 64KB TAGE-SC-L [68] predictor, depending on the component used for the prediction and the counter values. Fig. 6a shows that indeed, when the prediction uses the saturated counters of HitBank or of the bimodal predictor, the probability of a misprediction is close to zero. However, when there was a miss in the last eight predictions provided by the bimodal predictor (bimodal >1in8), the misprediction rate is higher than 6% on average, although the counters are actually saturated (-2 and 1). Fig. 6b shows a similar trend for the Statistical Corrector (SC), that is, the higher the absolute value of the output is, the higher the prediction confidence. Yet, the miss rate remains quite high (around 10%) even if the output value is saturated.

Fig. 6 is completed by Fig. 7, which illustrates the misprediction contribution of different components within TAGE-SC-L. One can observe that, on average for the traces used in this work, 66.7% of the mispredictions are provided by the HitBank. The AltBank account for 8.1% of the total mispredictions. The bimodal component incurs 6.2% when no misses are found in the last 8 predictions and 7.5% otherwise. The Loop Pedictor (LP) negligibly contributes to mispredictions (0.1%). SC accounts for 11.1% of the mispredictions.

Driven by the miss rates shown in Fig. 6, we improve the TAGE confidence estimation heuristic [67] in several ways. First, we underline that the original heuristic does not differentiate between predictions stemming from the HitBank or from the AltBank. In contrast, we analyze the confidence of the predictions per bank and show that predictions stemming from the AltBank *always* exhibit a very high miss rate,

regardless of the value of their counter, as shown in Fig. 6a. Hence, in this work, we consider that any prediction provided by AltBank has low confidence, which is noticeable, given its 8.1% fraction of the total mispredictions.

Second, since the original TAGE confidence estimation was developed for a simpler TAGE predictor, we extend in this work the confidence estimation to LP and SC. Fig. 6b shows a particularly low miss rate in predictions originating from LP in TAGE-SC-L (<3%, independently from the confidence value) and therefore consider LP predictions as high-confidence. On the other hand, the confidence of SC predictions in TAGE-SC-L vary depending on the absolute SC output value (Fig. 6b) from 10% to 50%, so they cannot be considered as high confidence. SC represents 11.1% of the total mispredictions. These extensions improve both the accuracy and coverage of the original TAGE confidence estimator and add support for LP and SC, as shown in Section VI, without any extra storage.

B. Initiating the Alternate Path

Using our described confidence estimator built on top of TAGE-SC-L, we classify a given branch instance as H2P if its prediction is from (1) bimodal if there was a misprediction in the past 8 branches predicted by bimodal. (2) bimodal or HitBank for which the prediction counter is not saturated, (3) AltBank, and (4) SC. Generally, this corresponds to predictor entries for which the misprediction rate is above 5%, according to Fig. 6. At branch prediction time, if a conditional branch is identified as H2P, alternate path generation is initiated.

C. Generating the Alternate Path

To generate alternate path addresses past a single basic block, an entire BPU is required, including a BTB, an indirect target predictor, a RAS, and a branch predictor. Replicating those structures to predict the alternate path would add considerable area overhead, since they are the largest frontend structures, e.g. 560KB for the BTBs and branch predictor [25].

Hence, we opt for doubling the number of banks (from 16 to 32) of our baseline banked BTB design [49], which are shared between the predicted path and the alternate path. This lets us retrieve branch targets on both the predicted and alternate paths without implementing a separate BTB, at the cost of bank conflicts. Practically, at the beginning of the BTB access cycle, we determine which banks need to be accessed by the predicted and alternate path. On a conflict, rather than selecting a winner that "takes all", accesses are resolved in the following way: UCP keeps a 3-bit saturated counter to track the number of cycles that the current alternate path PC has been delayed due to a conflict. When the counter saturates, the alternate path is allowed to win the conflicted banks, causing the demand path to retry in the next cycle. The counter resets when the current PC of the alternate path changes.

As banking incurs area and latency costs, other BTB organizations such as the region BTB (an entry covers n taken-at-least-once branches of an aligned code region) or block-based BTB (an entry covers a dynamic block of i instructions with at most n taken-at-least-once branches) could be considered [51]. With those, both paths would access a single entry, such that concurrent predictions could be achieved with only a handful of banks. However, since UCP is conceptually agnostic of the BTB organization, we only considered the instruction BTB.

For conditional branches, we use a small TAGE-SC-L branch predictor [69] (*Alt-BP*). The reason for building a dedicated conditional predictor on the alternate path is that naively banking the tagged tables by restricting each PC to a single bank within a tagged table significantly harms performance, and efficiently banking TAGE to enable multiple predictions per cycle has not been covered in the literature and is beyond the scope of this work. Alt-BP is updated along with the main branch predictor, meaning that its GHR will diverge from the predicted path only when alternate path is initiated. In practice, Alt-BP implements two GHRs. When alternate path starts, the predicted path GHR pre-H2P branch is copied into the alternate path GHR, and the two are speculatively updated with the predicted direction and its opposite, respectively. From that point on, the predicted path GHR of Alt-BP is speculatively updated with predictions from the main predictor, while the alternate path GHR is speculatively updated using predictions from Alt-BP, which are made using the alternate path GHR. When the alternate path exits, no specific care has to be taken, as the alternate path GHR will be resynchronized once a new alternate path starts again.

Operating Alt-BP in this fashion implies that its prediction tables are not updated if the alternate path is incorrect. Indeed, during alternate path operation, predictions are generated for the alternate path only. Therefore, if the predicted path is correct, there is no corresponding state captured in the FIFO structure used to update Alt-BP (i.e. entry number, counter value). However, updates on the alternate path are performed if the alternate path becomes correct, as a pipeline flush will take place and Alt-BP will eventually be updated with the corrected path information.

Our UCP proposal leverages a small ITTAGE [66] (*Alt-Ind*) indirect predictor to prevent early exiting the alternate path because an indirect branch target is unknown. We use a dedicated predictor for the same reason as the branch predictor: banking efficiency. It operates similarly to Alt-BP (GHR, updates). However, and as we will show in Section VI, the gains brought by a dedicated indirect target predictor are generally limited on average and UCP could be implemented without a dedicated indirect target predictor to limit overhead. Finally, to handle returns on the alternate path, we use a dedicated RAS (*Alt-RAS*). The main RAS is copied into the Alt-RAS when alternate path UCP starts, and it is updated speculatively when walking the alternate path.

Both main path and alternate path address generation are performed in parallel. The generated addresses from both paths are added to their respective FTQs (named *Alt-FTQ* for the alternate path).

TABLE I: Weights added to the saturation counter on specific events on the alternate path.

	Prediction Source	Predictor Output	Weight
Condition	BiModal	-2 & 1	1
		-1 & 0	2
	BiModal (>1in8)	-2 & 1	2
		-1 & 0	6
	HitBank	-4 & 3	1
		-3 & 2	3
		-2 & 1	4
		-1 & 0	6
	AltBank	-4 & 3	5
		-3, -2, -1, 0, 1, 2	7
	Loop Predictor	Any	1
	SC	128 to 255	3
		64 to 127	6
		32 to 63	8
		0 to 31	10
Target	BTB Miss	–	∞
	Indirect branch	–	1 (∞)
	Return branch	–	1

D. Prefetching the Generated Alternate Path

Addresses at the head of the Alt-FTQ are first used to perform a µ-op tag check before initiating a prefetch request, to prevent prefetching instructions already present in the µ-op cache. This tag check is conducted simultaneously with other ongoing tag checks on the predicted path, and is facilitated by set interleaving the µ-op cache into two 2-ported banks. In the event of a conflict during the tag check process, priority is given to the address on the predicted path, while the alternate path address attempts again in the next cycle, similar to the BTB accesses. Once the µ-op cache tag check completes, the address is removed from Alt-FTQ.

Upon a µ-op cache miss, a prefetch request for the cache line corresponding to the missing instruction is recorded in the µ-op cache Miss Status Holding Register (MSHR) and inserted in the L1I prefetch queue (PQ). From the PQ, it proceeds as a standard L1I prefetch: if the entry is not already present in L1I, the cache line will be fetched from L2, LLC, or memory. The system is able to process only one prefetch request per cycle, but since the L1I is set-interleaved, both demand and prefetch requests can proceed in the same cycle if they map to different banks. When a cache line returns whose retrieval was initiated by alternate path UCP, the requested instructions are directed to a dedicated decode queue where they are decoded and inserted in the µ-op cache.

E. Stopping the Alternate Path

The alternate path address generation stops automatically in the following two cases: (1) a new H2P branch is detected, and therefore a new alternate path is initiated; (2) the path being explored is considered too unlikely to become the correct path. The heuristic to stop the alternate path builds on the heuristic for estimating confidence of branches and additionally considers target predictions.

The stopping heuristic relies on a 6-bit saturated counter that is initialized to zero when alternate path prefetching is triggered. The counter is incremented with a different weight every time a branch is encountered on the alternate path. The weights are adjusted based on the average hit rate of each branch prediction category (Fig. 6) (approximately 1 unit per extra 5% miss rate – see Table I for details). The higher the value of the counter, the more unlikely for the next basic block in the alternate to become the correct path.

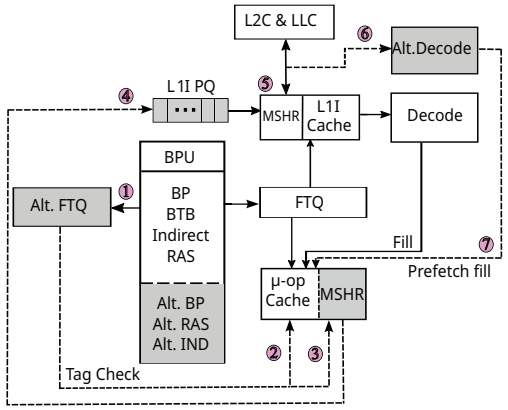


Fig. 8: New structures and data-paths required by UCP

The alternate path stops either when the counter reaches an established threshold (e.g., 500 in our work) or when a clear low confidence event (e.g. a BTB miss) occurs. We also stop on the detection of an indirect branch if we do not employ an Alt-Ind predictor. Finally, to avoid indefinitely generating addresses for loops never predicted to end, and to restrict it to the critical instructions, the threshold is incremented by 1 for high confidence branches. As the threshold is updated only when a predicted branch is encountered, UCP can continue generating prefetching addresses if no branches are found. To avoid this, UCP keeps a 6-bit counter that resets on each predicted branch and is incremented by 1 for each instruction on the alternate path. The alternate path will cease once the counter has reached its maximum value.

F. Overview and Hardware Overhead

Fig. 8 depicts the modifications required for UCP. Gray boxes indicate the added components and dotted lines represent newly introduced data paths. Our design incorporates an 8KB TAGE-SC-L branch predictor (Alt-BP), a 4KB ITTAGE indirect target predictor [66] (Alt-Ind), and a 16-entry Alt-RAS (0.06KB), that are combined to the BTB for generating alternate path addresses. These addresses populate an alternate 24-entry FTQ that holds µ-op cache entry addresses (0.14KB) ❶. A 32-entry µ-op cache MSHR (0.19KB) is also employed to monitor ongoing prefetch requests ❷. We double the tag check bandwidth to the µ-op cache by banking the µ-op cache and managing conflicts ❸, as in the BTB. Prefetches that miss the µ-op cache are inserted in the L1I Prefetch Queue (PQ, 0.25KB) ❹. After prefetch completion ❺, instructions enter a 32-entry alternate decode queue (0.12KB) and are subsequently decoded using 6 dedicated *Alt-Decoders* ❻ before being added to the µ-op cache ❼. The overall memory overhead required by UCP is 12.95KB (8.95KB when not leveraging an Alt-Ind predictor).

G. Other Concerns & Design Points

1) Impact of ISA on µ-op Cache Prefetching: The main concern regarding the ISA is whether instruction decoding is stateful or not. Consider the following example where a basic block starts at byte 16 in cache line A, and ends at position 61 in cache line A + 1. In x86, in which decoding is stateful (handling variable length instructions), decoding instructions in cache line A + 1 requires having decoded instructions in cache line A. In the context of µ-op cache prefetching, although we enqueue prefetches in program order, i.e. A then A + 1, it is possible that cache lines are retrieved from the memory hierarchy out-of-order, i.e., A + 1 then A. In this case, the decoding process has to either stall until A returns from memory, or, decode another younger line that starts a new basic block, if available. The latter approach runs the risk that when A returns from memory, the decoders are busy, and A misses its window to be decoded and inserted in the µ-op cache in a timely fashion. In contrast, in ARMv8, in which decoding is stateless (because instruction length is fixed and instructions are aligned) decoding may be performed out-of-order, as the cache lines return from the memory hierarchy.

2) L1I Inclusivity of µ-op Cache: Similarly to instruction and data caches, the µ-op cache may be indexed using virtual addresses or physical addresses. The former may have lower latency if the Instruction TLB (ITLB) is large and its latency cannot be fully hidden by the tag array access. Indeed, as long as the ITLB maintains inclusivity of the µ-op cache, the ITLB check can be bypassed as any hit on a virtual tag implies that the translation is valid. Aliasing needs to be handled, but invalidation of aliased lines can be achieved by invalidating the whole set as long as aliases reside in the same set, which is the case if set index bits are not translated. Alternatively, aliasing can be prevented by also keeping the L1I inclusive of the µ-op cache and preventing two entries of the same µ-op cache set from pointing to the same L1I way (assuming, again, that all aliases have to reside in the same set in the µ-op cache). This seems to be the approach favored by Intel [31] (256-entry ITLB in Alder Lake).

However, virtually tagging the µ-op cache has significant drawbacks, especially in processors implementing Simultaneous Multithreading (SMT), as code shared in the physical space of the L1I will create distinct entries in the virtual space of dynamically shared µ-op cache [40]. We speculate that this is the reason why Amd favors a physically tagged µ-op cache in its recent Zen 4 microarchitecture [8] (64-entry ITLB). A physically tagged µ-op cache does not need to be contained into the L1I or the ITLB, although doing so may facilitate invalidations caused by e.g., cache maintenance operations. For instance, if the geometries are similar (number of sets, ways, and bytes covered by an entry), invalidation requests need only search one structure and invalidate the matching set/way in both.

Nevertheless, keeping the µ-op cache included in the L1I and ITLB prevents from caching a larger portion of the code within the core, as the maximum amount of cached instructions is still limited by the size of the L1I. While this may be a worthy trade-off when the µ-op cache is virtually tagged if the ITLB lookup is costly, it does not appear advantageous if the µ-op cache is physically tagged, especially as cache line invalidations are not the common case.

In this work, we use a physically tagged µ-op cache that is not included in the L1I or ITLB, so as to maximize reach. Although a miss in L1I when performing alternate path UCP behaves much in the same way as it would in an inclusive L1I (a line is allocated in both structures), we are able to retain µ-op cache entries even if the corresponding L1I entry is evicted by the replacement policy.

V. Methodology

We evaluate UCP by integrating our modifications in the *develop* branch of ChampSim [4].[2] ChampSim includes a detailed frontend model implementing fetch-directed prefetching (FDP) [56], a branch target Buffer (BTB), indirect target predictor, return address stack (RAS), and conditional branch predictor. L1I prefetch requests issued through FDP are actually demand accesses and, therefore, we do not consider them as prefetch requests. That is, we assume a given address in FTQ checks the L1I tags a single time and fetches the instruction bytes, as opposed to checking it once for the purpose of prefetching and a second time when it reaches the head of the FTQ as a demand request.

We extend ChampSim's standard µ-op cache design to reflect the frontend described in Section II. Specifically, our frontend works either in *stream* mode or in *build* mode, paying a 1-cycle penalty when switching modes [57]. The µ-op cache entries follow all termination conditions discussed in Section III. The L1I is even/odd interleaved so that basic blocks spanning two cache lines can be retrieved in a single cycle. Interleaving also enables sharing the L1I tag lookup bandwidth between the predicted path and alternate path at no extra cost over the baseline. The baseline µ-op is dual ported, and its tag arrays are even/odd interleaved in UCP.

The processor and memory hierarchy are configured following the specifications of an Intel's latest Alder Lake performance core. The primary parameters are listed in Table II.

We evaluate our proposal using the Qualcomm Datacenter traces provided in the first Championship on Value Prediction (CVP-1) [1]. The traces [50], which also include the 50 IPC1 traces [2], have been converted to ChampSim format using the most recent converter [20]. Our analysis considers the 306 (out of 2011) CVP-1 traces (2 FP, 97 INT, 73 Crypto and 134 datacenter traces) that show at least a 5% IPC improvement over our baseline configuration when using an ideal µ-op cache. We execute 100 million instructions, the first 50 million used for warm-up and the next 50 million used to collect statistics. We use the geometric mean for speedups, and the arithmetic mean for other metrics.

[2]Commit `c8eff1dafdb398fcb9a40c95994cb202d831d678`

TABLE II: Baseline configuration

Out-of-order processor	
Branch prediction	64K-entry 16-bank instruction BTB [51] LRU, 64KB ITTAGE [66], 64-entry RAS, 64KB TAGE-SC-L [68]
µ-op cache	4Kops, 64 sets, 8 ways, 8 µ-op/entry, 1-cycle hit, LRU [40], [43], 2 ports
Frontend Stages	Up to 16 sequential addresses predicted per cycle, 16-wide fetch, 6-wide Decode, 6-wide Dispatch, 192-entry FTQ, 32-entry decode buffer, 32-entry dispatch buffer
Backend Stages	10-wide Execute, 3x load, 2x stores, 10-wide Commit, 512-entry ROB, 192-entry LQ, 114-entry SB
Memory hierarchy	
ITLB	256 entries, 8 ways, 1-cycle hit, 8-entry MSHR
DTLB	96 entries, 6 ways, 1-cycle hit, 8-entry MSHR
STLB	2048 entries, 16 ways, 8-cycle hit, 16-entry MSHR
L1I	32KB, 8 ways, 4-cycle hit, 32-entry PQ, 16-entry MSHR, LRU, 2 banks
L1D	48KB, 12 ways, 5-cycle hit, 16-entry MSHR, IP-stride prefetcher, 8-entry PQ, LRU
L2	1.25MB, 20 ways, 10-cycle hit, 32-entry MSHR, LRU
LLC	30MB, 12 ways, 40-cycle hit, 64-entry MSHR, LRU
DRAM	2-channel, 8-bank, t_{RP}: 12.5ns, t_{RCD}: 12.5ns, t_{CAS}: 12.5ns

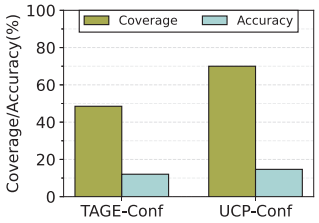


Fig. 9: Coverage and accuracy of H2P predictor

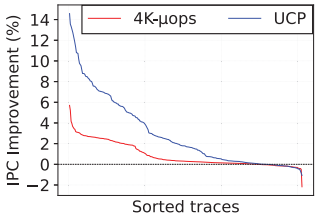


Fig. 10: IPC of UCP and baseline relative to no µ-op cache

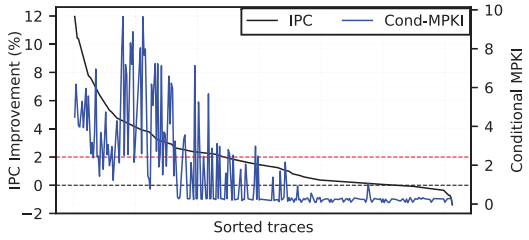


Fig. 11: Speedup and conditional branch MPKI

VI. Results

A. Coverage and accuracy of detecting H2P branches

Fig. 9 displays the coverage (indicating how many conditional branches mispredicts are marked as H2P) and accuracy (illustrating how many H2P branches actually mispredict) of TAGE-Conf and UCP-Conf respectively, as H2P predictors. As detailed in section IV-A, UCP extends the TAGE-Conf design to include SC, LP and AltBank. This enhancement enables UCP-Conf to improve coverage from 48.5% to 70%. Additionally, since both SC and AltBank exhibit high miss rates, accuracy also improves from 12% in TAGE-Conf to 14.66% in UCP-Conf.

B. Performance

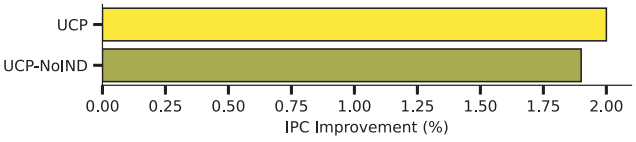
Fig. 11 demonstrates how UCP improves performance compared to the baseline across diverse applications along the conditional branch MPKI. Fig. 11 shows UCP improvement over baseline (Table II), while Fig. 10 presents the improvement of both UCP and the baseline over a configuration without a µ-op cache. With UCP, 90% of the applications used in this study benefit from a µ-op cache, compared to 80.7% in the baseline, while the remaining 10% show negligible performance degradation (<0.8%).

On average, performance is increased by 2%, up to 12%, with an average MPKI of 1.56 and 6.17 respectively, for the workload benefiting the most from UCP. Although a higher MPKI does not guarantee speedup, it generally entails it, confirming that it is beneficial to ensure that the µ-op cache contains the µ-ops that follow a misprediction, so that the pipeline can be swiftly refilled.

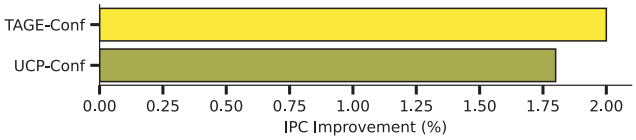
However, UCP is occasionally detrimental to performance, as shown in Fig. 11. One major reason for this degradation is higher fetch pipeline switch frequency. For example, in application *srv207*, the switches PKI increase from 9.8 to 10.2, yielding a 0.5% slowdown. Switches PKI can increase because although we improve likelihood for seeing a long stream of µ-op cache hits after a branch misprediction, the µ-ops brought by UCP displace existing entries. This can split a long stream of hits into multiple smaller streams, incurring the switch penalty more often. If this penalty is not hidden, performance decreases.

We have built two UCP flavours, with and without a dedicated indirect predictor, trading storage for lookahead potential. We observed that, in the absence of an indirect predictor, if the alternate path is correct, approximately 33.7% of the generated paths are halted. We therefore experiment with adding a 4KB ITTAGE [66] to act as the alternate path indirect predictor. The IPC improvement is depicted in Fig. 12a along the baseline UCP configuration: a dedicated alternate indirect predictor further pushes the speedup from 1.9% to 2%. The maximum benefit stands at 10.6% when not using a dedicated indirect predictor and 12% when using a 4KB ITTAGE. Similarly minimum benefit is -1.4% without indirect predictor and -1.3% when using an indirect predictor.

As UCP relies on information from the branch predictor to initiate the alternate path, having a confidence mechanism that offers high coverage and accuracy is crucial for UCP. Fig. 12b illustrates the IPC improvement when using Seznec's TAGE-based confidence mechanism [67] compared to the improved version we use in UCP. As detailed in section IV-A, the updated mechanism achieves superior accuracy and coverage, leading to an 10% additional IPC improvement compared to TAGE-conf (1.8% vs. 2% average speedup). TAGE-Conf achieves a maximum speedup of 11.6%, while UCP-Conf reaches a maximum speedup of 12%. Similarly, the minimum benefits stand at -2.6% and -1.3%, respectively.



(a) UCP with and without indirect predictor



(b) TAGE confidence estimation vs. UCP (with indirect predictor)

Fig. 12: Speedup of different configurations

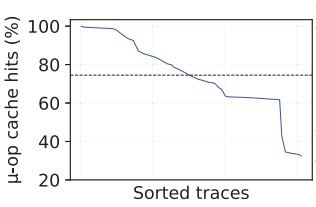


Fig. 13: µ-op cache hit rate

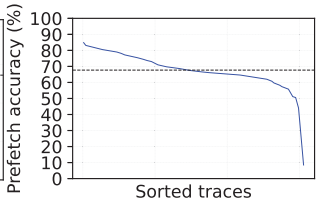


Fig. 14: Prefetch accuracy

C. µ-op cache hit rate

Fig. 13 illustrates the µ-op cache hit rate for UCP. As expected, on average, the UCP µ-op cache hit rate shows little improvement, from 71.4% to 74%, as our prefetching strategy is directed towards very few, but critical instructions. On average, UCP prefetches as little as ten cache lines per alternate path.

D. Prefetch accuracy

Fig. 14 displays the prefetch accuracy of UCP, calculated as the number of timely prefetches over the total number of prefetches, at the µ-op cache entry granularity. On average, the accuracy is 67.7%. Note that prefetches are considered timely with respect to the current instance of the target H2P branch. However, if the H2P branch was correctly predicted and the alternate-path is not useful for the current instance, once the µ-ops are cached, they are likely to be useful for future instances of the same H2P branch, as they are likely to mispredict. In our study we found that 8% (maximum 18%) of the prefetched µ-op cache entries are used at least once even if they were prefetched on an incorrect alternate path. These cases are not accounted for in the computation of the accuracy.

E. Stopping Threshold Sensitivity Analysis

Fig. 15 depicts the improvements in IPC across various threshold values (ranging between 10 to 10000) used to terminate alternate path for prefetching in the µ-op cache and prefetching only until the L1I (UCP-L1I). For prefetching till µ-op cache we observe that applications where branches on the alternate path are comparatively easy to predict tend to perform better with higher thresholds, thanks to longer correct alternate paths (e.g. 21.5 cache lines on average per

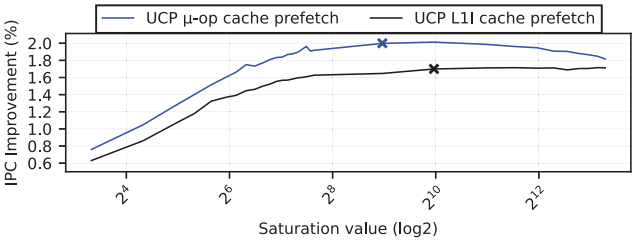


Fig. 15: Sensitivity analysis of average IPC for various saturation values under.

alternate path in *srv203*, compared to the 9.9 lines average across all applications). Conversely, some applications benefit from lower threshold values, because of a higher number of low-confidence predictions on the alternate path. A smaller threshold allows prefetching to conclude early, preventing the µ-op cache from being cluttered with µ-ops that will not be immediately required if the predicted path proves incorrect. However, the trend across traces is that for threshold values higher than 500, the IPC improvement reaches a plateau. This is due to the fact that the majority of the speedup is derived by prefetching the µ-ops immediately following a branch misprediction. Once the pipeline is populated again, the benefits of prefetching longer alternate paths diminish. However, if the threshold is increased beyond a certain limit (around 1000 in our study), µ-op cache thrashing is observed, as shown in Fig. 15.

Prefetching only until L1I, alleviates the need for dedicated decoders and results in a performance improvement in the range of 0.6% to 1.7%. The highest IPC improvement is obtained at a threshold value of 1000 (an improvement of 1.6% is observed at the same threshold as UCP). This threshold is distinct from that of the µ-op cache, as L1I is typically larger than µ-op cache, and thus the cache thrashing effect is observed at a higher threshold. Without being the main goal of UPC, using only L1I prefetching with a threshold of 1000 outperforms several previously proposed state-of-the-art L1I prefetchers, delivering a 1.7% improvement with relatively low hardware overhead. UCP provides 2% when prefetching till the µ-op cache. Nonetheless, this leaves part of the potential untapped. Ideal branch recovery up to the next 8 (resp. 16) branches can provide improvements of 2.3% (resp. 2.9%), as discussed in Section III.

F. Cost/Benefit analysis

Performance improvement is typically achieved with an increase in hardware complexity. Thus, we compare UCP to other frontend techniques (L1I prefetchers, µ-op caches, Misprediction Recovery Cache or MRC [48], and an improved branch predictor) in terms of speedup per invested hardware. The storage cost/benefit analysis is plotted in Fig. 16. Our analysis highlights that both UCP flavours (8KB TAGE-SC-L with and without a 4KB ITTAGE indirect branch predictor for the alternate path) are on the Pareto front, i.e. provide the most improvement per KB of storage. In terms of absolute IPC,

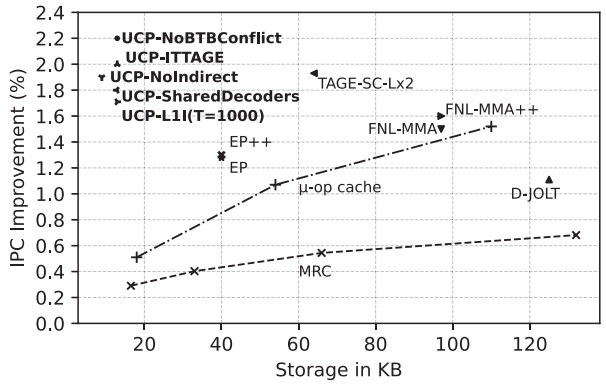


Fig. 16: Performance improvement and storage requirement of UCP, L1I prefetchers (EP, EP++, DJOLT, FNL-MMA, and FNL-MMA++), µ-op caches (8Kops, 16Kops, 32Kops), MRC using several sizes, and a 128KB TAGE-SC-L

doubling the size of the branch predictor achieves marginal improvement over UCP, at multiple times the cost.

We also implemented MRC as a fully associative cache with an LRU replacement policy. Each MRC entry stores a tag, 64 µ-ops per entry, and replacement information. On a branch misprediction, an MRC entry is allocated and records the 64 µ-ops following the corrected path. It is tagged with the correct branch target. The MRC is thus able to stream up to 64 µ-ops on a misprediction, on a tag match, thus accelerating pipeline refill. We tested MRC sizes of 16.5KB, 33KB, 66KB, and 132KB, yielding IPC improvements of 0.3%, 0.4%, 0.55%, and 0.7% respectively. This is because MRC records a single trace among the many possible for each conditional branch, while UPC can generate address traces on the fly through the BTB and branch predictors.

In addition, we tested sharing the 6 baseline decoders between the two paths, to determine the impact of not implementing dedicated decoders. Sharing is "winner takes all", that is, the alternate path can decode only when the predicted path is hitting in the µ-op cache (in streaming mode). Doing so (UCP-SharedDecoders) allows UCP to provide a geomean improvement of 1.8% (vs. 2% when using dedicated decoders). Regardless, we argue that dedicated decoders will have only moderate impact on the dynamic energy consumption as UCP increases the number of decoded instructions by merely 25.5% (Section VI-D).

Lastly, UCP has been tested in an ideal BTB banking environment, meaning there cannot be BTB conflicts between PCs on the alternate and demand paths. Our study shows that in an ideal BTB setting (UCP-NoBTBConflict), UCP has the potential to further increase the IPC improvement from 2% to 2.2%, on average.

VII. Related work

While a plethora of L1I prefetchers have been proposed, none were considered to tackle µ-op cache prefetching, to the best of our knowledge. Prefetching the alternate-path resembles prior techniques that exploit wrong-path execution, although our technique only adds lightweight hardware as it does not speculatively allocate backend resources to wrong-path instructions, to be reused upon a misprediction. This section briefly overviews standalone L1I prefetchers and fast pipeline refills, then delves in techniques to leverage the alternate-path, and finally in prior work on µ-op cache management.

A. Standalone L1I Prefetchers

Recent years have seen many standalone L1I prefetcher proposals [9], [24], [27], [37], [45], [47], [59], [70]. Both L1I and µ-op cache prefetching are designed to complement FDP, that is limited to fetching instructions only from the predicted path. In contrast, we rely on the existing BTB and small predictors to generate another path to follow, whereas standalone prefetchers can require up to 125KB (D-JOLT) to provide lower IPC improvements than UCP.

B. Fast Pipeline Refill

Grayson *et al.* introduces the Mispredict Recovery Buffer (MRB) in the Samsung Exynos 5 microarchitecture [25]. The MRB contains the addresses of the next three basic blocks following low confidence branches, and can provide the addresses much faster than the main BTB, hastening refills. Perais *et al.* proposes Elastic Instruction Fetching (ELF) [52], aiming to hide the increased misprediction penalty of decoupled fetching. ELF enables the fetch stage run ahead of the branch prediction stage on pipeline restarts, using a dedicated branch predictor. As these schemes relate to fetch address generation, rather than instruction retrieval, and do not target the µ-op cache, they are orthogonal to our work.

The Mispredict Recovery Cache (MRC) [48] stores streams of decoded instructions following branch mispredictions. On a misprediction, the MRC is accessed, and if there is a hit, the cached instructions are fed to the execution unit directly, averting the penalty of refilling the entire pipeline. This work is the most relevant to our proposal as it explicitly focuses on caching decoded instructions on alternate paths. However, the MRC only participates in frontend operation following a misprediction, whereas a µ-op cache can provide µ-ops at any time. Moreover, the MRC should likely grow with the code size to remain beneficial. Conversely, our proposal leverages decoupled fetching to prefetch as needed. Lastly, the MRC is akin to a trace cache [62] and is more complex to maintain coherent than the µ-op cache.

C. Leveraging the Alternate Path

We distinguish two types of techniques that rely on the alternate path: control-flow reconvergence (CFR) and multi-path operation (MPO).

CFR preserves instructions that are not dependent on the branch, once the branch is found to be mispredicted. Those instructions are executed on either path, and do not conceptually need to be re-fetched, re-dispatched and re-executed. CFR requires logic to identify the reconvergence

point [7], [18], [22], [42], [53], [63] and is orthogonal to µ-op prefetching as it does not fetch the pre-reconvergence point alternate path until the misprediction is detected.

MPO techniques process both taken and not-taken paths concurrently. They competitively fetch, allocate [11], and execute instructions [41], [74], [76]–[78], from both paths, such that, upon a misprediction, some correct control-flow instructions have certainly been executed. While processing two or more paths reduces branch recovery time, it comes at the cost of either (1) significant hardware complexity, if additional resources are added to absorb the processing of alternate paths, or (2) running the risk of decreasing performance if alternate paths competition for pipeline resources is not well managed. Moreover, pipeline resources allocated to incorrect path eventually need to be scrubbed. Conversely, UCP does not allocate pipeline resources to alternate path instructions, except for µ-op cache entries. As such, the competition between the two paths is kept minimal, and terminating alternate path µ-op cache prefetching only requires flushing the A-FTQ. Akin to prior work, our proposal also requires additional hardware, as we introduce dedicated decoders and an alternate branch predictor, but the complexity and overhead is comparatively limited (8.95 to 12.95KB).

D. Identifying Hard to Predict Branches

Alternate path processing requires confidence information to filter which branches trigger alternate path processing. This is achieved either through dedicated structures [6], [10], [26], [33], [67], [75] or using information readily available in the branch predictor [6], [67], [72].

Jacobsen *et al.* predict whether a branch has low confidence by monitoring the outcome history (correct/incorrect) of branches *xor* global branch histories, e.g. if there are more incorrect than correct outcomes in the selected history vector, then the branch is hard to predict. This scheme is used in *Polypath* to drive alternate path execution [41] and improved on by Grunwald *et al.* [26].

Tyson *et al.* [75] build a set of global branch histories that correspond to high confidence branches through profiling. At runtime, if the current global history is found in the set, the branch is confident, otherwise alternate path fetching starts. Pruett *et al.* [54] introduce a Hard Branch Table (HBT) for identifying H2P branches. Each entry, allocated upon the retirement of a conditional branch, includes a 5-bit saturating misprediction counter whose saturation indicates an H2P branch. Similarly, Gao *et al.* [23] introduce the H2P Branch Tracking Table (HBTT) which is a cache-like structure that tracks the misprediction and increment a saturation counter when the miss occurs. Finally, Aragón *et al.* [10] propose the Branch Prediction Reversal Unit (BPRU), which relies on data value correlation to identify low confidence branches. However, its overhead is prohibitive as the BPRU is larger than the branch predictor itself. In general, all these approaches may face challenges with datacenter traces, as the tables tracking hard-to-predict branches are typically small and may be overwhelmed by large instruction footprints.

Conversely, Seznec [67] and Akkary *et al.* [6] leverage the value of prediction counters of the TAGE and Perceptron branch predictors respectively, to estimate branch prediction confidence. Our confidence estimator is largely based on Seznec [67], with improvements such as considering which table provides the prediction.

E. µ-op Cache Management

A number of previous studies have attempted to increase µ-op cache efficiency by relaxing entry termination rules. Kotra *et al.* [43] identifies heavy fragmentation in µ-op cache and mitigate it through dedicated optimizations: Cache Line boundary AgnoStic µ-op cache design (CLASP) and µ-op cache compaction. In the same spirit, Moody *et al.* [46] introduces Speculative Code Compaction (SCC), a microarchitectural technique that speculatively eliminates dead instructions from hot code regions in the µ-op cache. Kim *et al.* [40] identifies significant congestion in the µ-op cache of a simultaneous multithreading processor employing competitively shared µ-op caches. They propose a software-based logical partitioning of the µ-op cache that relies on a just-in-time compiler to rearrange the code of competing threads. Yet, this is not applicable to multiprogrammed workloads where native binaries execute concurrently, as they cannot easily be re-arranged to improve µ-op cache utilization. In a nutshell, prior techniques for µ-op cache management improve the utilization of the µ-op cache, but do not focus on reducing the processor stalls caused by critical instructions, as is the case for UCP.

VIII. Conclusion

In this paper we show that by targeting code that Fetch Directed Prefetching cannot prefetch by construction and by focusing on few but critical instructions, significant performance benefits can be obtained. Specifically, we prefetch in the µ-op cache only a few cache lines worth of the alternate path of hard-to-predict conditional branches, achieving an average of 2% and up to 12% speedup (resp. 1.9% and up to 10.6%) with a moderate storage overhead of 12.95KB (resp. 8.95KB), which includes alternate predictors and queues to track and follow the alternate path. Our technique outperforms larger µ-op caches or prefetching in the µ-op cache using a standalone L1I prefetcher.

Acknowledgment

This work was supported by the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement No 819134), the MCIN/AEI/10.13039/501100011033/ and the "ERDF A way of making Europe", EU (grant PID2022-136315OB-I00), the MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU/PRTR (grant TED2021-130233B-C33), the European Union's Horizon 2021 research and innovation program (grant agreement No 101070374 under HORIZON-CL4-2021-DIGITAL-EMERGING-01) and the Ramón y Cajal Research Contract (RYC2018-025200-I).

REFERENCES

[1] "The First Championship Value Prediction (CVP1)," Jun. 2018. [Online]. Available: https://www.microarch.org/cvp1/cvp1/index.htm

[2] "The 1st instruction prefetching championship (IPC1)," May 2020. [Online]. Available: https://research.ece.ncsu.edu/ipc/

[3] "Amd's zen 4 part 1: Frontend and execution engine," Nov. 2022. [Online]. Available: https://chipsandcheese.com/2022/11/05/amds-zen-4-part-1-frontend-and-execution-engine/

[4] "ChampSim simulator, develop branch," Nov. 2022. [Online]. Available: https://github.com/ChampSim/ChampSim/tree/develop

[5] Advanced Micro Devices, "Software Optimization Guide for AMD EPYC™ 7003 Processors, Pub 56665, Rev 3," p. 33, [Online; accessed Aug.-2023].

[6] H. Akkary, S. T. Srinivasan, R. Koltur, Y. Patil, and W. Refaai, "Perceptron-based branch confidence estimation," in *10th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2004, pp. 265–265.

[7] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary, "Transparent control independence (tci)," in *35th Int'l Symp. on Computer Architecture (ISCA)*, 2007, pp. 470–481.

[8] AMD, "Software Optimization Guide for AMD Zen4 Microarchitecture, Pub 57647, Rev 1, section 2.9.1," p. 21, Jan. 2023, [Online; accessed Nov.-2023].

[9] A. Ansari, F. Golshan, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Mana: Microarchitecting an instruction prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[10] J. L. Aragón, J. González, J. M. Garcia, and A. González, "Confidence estimation for branch prediction reversal," in *8th International Conference on High Performance Computing*. Springer, 2001, pp. 214–223.

[11] J. L. Aragón, J. González, A. González, and J. E. Smith, "Dual path instruction processing," in *16th Int'l Conf. on Supercomputing (ICS)*, 2002, pp. 220–229.

[12] T. Asheim, B. Grot, and R. Kumar, "A storage-effective btb organization for servers," in *29th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 1153–1167.

[13] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. J. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," in *46th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2019, pp. 462–473.

[14] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 399–411.

[15] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, "The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v," *arXiv preprint arXiv:1607.02318*, 2016.

[16] G. Chacon, E. Garza, A. Jimborean, A. Ros, P. V. Gratz, D. A. Jiménez, and S. Mirbagher-Ajorpaz, "Composite instruction prefetching," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 471–478.

[17] A. Chauhan, J. Gaur, Z. Sperber, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Auto-predication of critical branches," in *47th Int'l Symp. on Computer Architecture (ISCA)*. IEEE, 2020, pp. 92–104.

[18] C.-Y. Cher and T. Vijaykumar, "Skipper: a microarchitecture for exploiting control-flow independence," in *34th Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 2001, pp. 4–15.

[19] M. Evers, L. Barnes, and M. Clark, "The amd next-generation "zen 3" core," *IEEE Micro*, vol. 42, no. 3, pp. 7–12, 2022.

[20] J. Feliu, A. Perais, D. A. Jiménez, and A. Ros, "Rebasing microarchitectural research with industry traces," in *Int'l Symp. on Workload Characterization (IISWC)*, 2023, pp. 100–114.

[21] M. Ferdman, A. Adileh, Y. O. Koçberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *17th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2012, pp. 37–48.

[22] A. Gandhi, H. Akkary, and S. T. Srinivasan, "Reducing branch misprediction penalty via selective branch recovery," in *10th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2004, pp. 254–264.

[23] H. Gao, Y. Ma, M. Dimitrov, and H. Zhou, "Address-branch correlation: A novel locality for long-latency hard-to-predict branches," in *14th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2008, pp. 74–85.

[24] N. Gober, G. Chacon, D. A. Jiménez, and P. Gratz, "Temporal ancestry prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[25] B. Grayson, J. Rupley, G. D. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the samsung exynos cpu microarchitecture," in *47th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2020, pp. 40–51.

[26] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence estimation for speculation control," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 122–131, 1998.

[27] V. Gupta, N. S. Kalani, and B. Panda, "Run-jump-run: Bouquet of instruction pointer jumpers for high performance instruction prefetching," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[28] V. Gupta and B. Panda, "Micro btb: A high performance and storage efficient last-level branch target buffer for servers," in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, 2022, pp. 12–20.

[29] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi, "Database servers on chip multiprocessors: Limitations and opportunities," in *Proceedings of the Biennial Conference on Innovative Data Systems Research*, 2007, pp. 79–87.

[30] A. Inc., "Arm® Neoverse™ V2 Core Technical Reference Manual, revision r0p2, Section 3.1," p. 68, Dec. 2022, [Online; accessed Nov.-2023].

[31] Intel, "Intel® 64 and ia-32 architectures optimization reference manual, section e.2.2.2," Jan. 2023, [Online; accessed Nov.-2023].

[32] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Re-establishing fetch-directed instruction prefetching," in *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2021, pp. 172–182.

[33] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *29th Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 1996, pp. 142–152.

[34] D. A. Jiménez, "Piecewise linear branch prediction," in *32nd Int'l Symp. on Computer Architecture (ISCA)*. IEEE, 2005, pp. 382–393.

[35] ——, "Multiperspective perceptron predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[36] D. A. Jiménez and C. Lin, "Dynamic branch prediction with perceptrons," in *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Jan. 2001, pp. 197–206.

[37] D. A. Jiménez, G. Chacon, N. Gober, and P. V. Gratz, "Barça: Branch agnostic region searching algorithm," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[38] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *42nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2015, pp. 158–169.

[39] 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 *54th Int'l Symp. on Microarchitecture (MICRO)*, 2021, pp. 816–829.

[40] J. Kim, H. Jang, H. Lee, S. Lee, and J. Kim, "Uc-check: Characterizing micro-operation caches in x86 processors and implications in security and performance," in *54th Int'l Symp. on Microarchitecture (MICRO)*, 2021, pp. 550–564.

[41] A. Klauser, A. Paithankar, and D. Grunwald, "Selective eager execution on the polypath architecture," in *25th Int'l Symp. on Computer Architecture (ISCA)*. IEEE Computer Society, 1998, pp. 0250–0250.

[42] V. R. Kothinti Naresh, R. Sheikh, A. Perais, and H. W. Cain, "Spf: Selective pipeline flush," in *36th Int'l Conf. on Computer Design (ICCD)*, 2018, pp. 152–155.

[43] J. B. Kotra and J. Kalamatianos, "Improving the utilization of micro-operation caches in x86 processors," in *53rd Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 2020, pp. 160–172.

[44] P. Michaud, "An alternative TAGE-like conditional branch predictor," *ACM Transactions on Architecture and Code onOptimization (TACO)*, vol. 15, no. 3, pp. 30:1–30:24, 2018.

[45] ——, "Pips: Prefetching instructions with probabilistic scouts," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[46] L. Moody, W. Qi, A. Sharifi, L. Berry, J. Rudek, J. Gaur, J. Parkhurst, S. Subramoney, K. Skadron, and A. Venkat, "Speculative code compaction: Eliminating dead code via speculative microcode transformations," in *55th Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 2022, pp. 162–180.

[47] 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)*, May 2020.

[48] A. K. Nanda, J. O. Bondi, and S. Dutta, "The misprediction recovery cache," *International journal of parallel programming*, vol. 26, no. 4, pp. 383–415, 1998.

[49] D. Parikh, K. Skadron, Y. Zhang, and M. Stan, "Power-aware branch prediction: Characterization and design," *IEEE Transactions on Computers*, vol. 53, no. 2, pp. 168–186, 2004.

[50] A. Perais, "1^{st} Championship Value Prediction Secret Traces," Jun. 2018. [Online]. Available: https://doi.org/10.18709/perscido.2023.02.ds384

[51] A. Perais and R. Sheikh, "Branch target buffer organizations," in *56th Int'l Symp. on Microarchitecture (MICRO)*. New York, NY, USA: Association for Computing Machinery, 2023, p. 240–253.

[52] A. Perais, R. Sheikh, L. Yen, M. McIlvaine, and R. D. Clancy, "Elastic instruction fetching," in *25th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2019, pp. 478–490.

[53] N. Premillieu and A. Seznec, "Syrant: Symmetric resource allocation on not-taken and taken paths," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–20, 2012.

[54] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in *54th Int'l Symp. on Microarchitecture (MICRO)*, 2021, pp. 804–815.

[55] G. Reinman, T. M. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *26th Int'l Symp. on Computer Architecture (ISCA)*, May 1999, pp. 234–245.

[56] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *32nd Int'l Symp. on Microarchitecture (MICRO)*, Dec. 1999, pp. 16–27.

[57] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead μops: Leaking secrets via intel/amd micro-op caches," in *48th Int'l Symp. on Computer Architecture (ISCA)*. IEEE, 2021, pp. 361–374.

[58] A. Ros and A. Jimborean, "The entangling instruction prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[59] ——, "A cost-effective entangling prefetcher for instructions," in *48th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2021, pp. 99–111.

[60] ——, "Wrong-path-aware entangling instruction prefetcher," *IEEE Transactions on Computers*, vol. 73, no. 02, pp. 548–559, 2024.

[61] E. Rotem, A. Yoaz, L. Rappoport, S. J. Robinson, J. Y. Mandelblat, A. Gihon, E. Weissmann, R. Chabukswar, V. Basin, R. Fenger, M. Gupta, and A. Yasin, "Intel Alder Lake CPU architectures," *IEEE Micro*, vol. 42, no. 3, pp. 13–19, Mar. 2022.

[62] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *29th Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 1996, pp. 24–34.

[63] E. Rotenberg, Q. Jacobson, and J. Smith, "A study of control independence in superscalar processors," in *5th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 1999, pp. 115–124.

[64] A. Saporito, "The ibm z15 processor chip set," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, 2020, pp. 1–17.

[65] A. Seznec, "The L-TAGE branch predictor," *The Journal of Instruction-Level Parallelism*, vol. 9, pp. 1–13, May 2007.

[66] ——, "A 64-Kbytes ITTAGE indirect branch predictor," in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, Jun. 2011.

[67] ——, "Storage free confidence estimation for the tage branch predictor," in *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*. IEEE, 2011, pp. 443–454.

[68] ——, "TAGE-SC-L branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Jun. 2016.

[69] ——, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[70] ——, "The fnl+mma instruction cache prefetcher," in *The 1st Instruction Prefetching Championship (IPC1)*, May 2020.

[71] A. Seznec, J. S. Miguel, and J. Albericio, "The inner most loop iteration counter: a new dimension in branch history," in *48th Int'l Symp. on Microarchitecture (MICRO)*, 2015, pp. 347–357.

[72] J. E. Smith, "A study of branch prediction strategies," in *8th Int'l Symp. on Computer Architecture (ISCA)*, 1981, p. 135–148.

[73] B. Solomon, A. Mendelson, D. Orenstein, Y. Almog, and R. Ronen, "Micro-operation cache: a power aware frontend for the variable instruction length isa," in *Proceedings of the 2001 international symposium on Low power electronics and design*, 2001, pp. 4–9.

[74] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 392–403.

[75] G. Tyson, K. Lick, and M. Farrens, "Limited dual path execution," Technical Report CSE-TR 346-97, University of Michigan, Tech. Rep., 1997.

[76] A. K. Uht, V. Sindagi, and K. Hall, "Disjoint eager execution: An optimal form of speculative execution," in *28th Int'l Symp. on Microarchitecture (MICRO)*. IEEE, 1995, pp. 313–325.

[77] A. Venkit, "Re-examining dual path processing," Ph.D. dissertation, 2022.

[78] S. Wallace, B. Calder, and D. M. Tullsen, "Threaded multiple path execution," *ACM SIGARCH Computer Architecture News*, vol. 26, no. 3, pp. 238–249, 1998.

APPENDIX

A. Abstract

The artifact contains a Docker image that launches experiments and collects results. The image executes the set of traces used to evaluate *UCP* and presents the IPC improvement of different *UCP* configurations discussed in the paper. It also reports *UCP Conf*'s coverage and accuracy for hard-to-predict branches.

B. Artifact check-list (meta-information)

- **Program:** Docker
- **Compilation:** Docker
- **Data set:** Subset of CVP1 traces provided within the docker image
- **Run-time environment:** Almalinux 9 (docker image can be run from any environment that supports docker)
- **Hardware:** Minimum requirement is a x86 machine with 16 GB memory. The suggested requirement for running the artifact in a reasonable time is a x86 cluster with 128 cores and 256 GB of memory.
- **Metrics:** IPC improvement and H2P accuracy and coverage
- **Output:** Table with final IPC improvement in percentage and H2P accuracy and coverage in percentage. IPC improvement graph is also generated.
- **Experiments:** *UCP*, *UCP-IdealBTBBanking*, *UCP-SharedDecoders*, *UCP-TillL1I*, and H2P accuracy and coverage. All testes are done at the threshold value of *500*.
- **How much disk space required (approximately)?:** 10GB
- **How much time is needed to prepare workflow (approximately)?:** 10 min
- **How much time is needed to complete experiments (approximately)?:** Sequential execution will take approximately 1 week, in parallel (128 cores) about 1 day
- **Publicly available?:** Simulation source code on Zenodo (DOI provided) and artifact image on DockerHub.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.10891466
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International, except for third-party codes.

C. Description

1) How to access: The Docker image is available at https://hub.docker.com/repository/docker/sawansingh/ucp_isca24/general and has been tested with docker version 24.0.5. For more details, please refer to the '*Repository Overview*' section of the link which contains a detailed guide to launch and verify the results.

2) Software dependencies: Our artifact only requires Docker (tested on version 24.0.5). With Docker available, the image will handle all necessary dependencies automatically.

3) Data sets: The evaluation of *UCP* is performed using a subset of traces from CVP1, as described in the section V. Note that the required traces are already included in the docker image.

D. Installation & running the artifact

Users can choose from two variants of the Docker image. One variant is configured to run all components automatically and print the results at the end of the process. The other variant allows users to enter the terminal of the artifact, check the files, and then run the main script. Please refer to the following guidelines to execute both variants.

Automatic, can be configured to run experiments in parallel or series. When running in parallel it takes the parameter to throttle the number of parallel jobs. To get started with the parallel version please run the following.

```
$ sudo docker pull
 ↪  sawansingh/ucpisca24:automated
$ sudo docker run -e "RUN_TYPE=parallel"
 ↪  -e "NUM_JOBS=8"
 ↪  sawansingh/ucp_isca24:automated
```

The *RUN_TYPE* option allows the user to select between serial and parallel execution. The *NUM_JOBS* parameter determines the number of jobs to be run in parallel. It is recommended that *NUM_JOBS* be set to a value that is less than or equal to the number of physical cores available on the machine.

To execute the experiments serially, the following command may be used. Please note that when executing serially, the *NUM_JOBS* argument is not required.

```
$ sudo docker run -e "RUN_TYPE=serial"
 ↪  sawansingh/ucp_isca24
```

Manual, allows the users to enter the docker terminal, run the scripts manually, and check the files. To run the manual mode, follow the following steps.

```
$ sudo docker pull
 ↪  sawansingh/ucpisca24:manual
$ sudo docker run -d --name ucp
 ↪  sawansingh/ucp_isca24:manual
$ sudo docker exec -it ucp /bin/bash
```

You should then be able to enter the container and see all the scripts and files by using the *ls* command. The main script that does all the work is named *run.sh*. To run the simulations you need to set two variables, *RUN_TYPE* and *NUM_JOBS*.

```
$ export RUN_TYPE=parallel
$ export NUM_JOBS=8
```

Then run *./run.sh* and you should see the following information

```
$ Welcome to UCP artifact! Run type:
 ↪  parallel, number of jobs: 8 (number
 ↪  of cores: 8)
```

After that, it will follow the same execution as the automated version.

E. Additional information

1) Copying files from docker to host: You can use the *docker cp* command to copy any file you need from docker to your machine.

For example, to copy the UCP binary from */champsim/bin/* to your directory, use

```
$ sudo docker cp ucp:/champsim/bin/UCP
 ↪  /path/to/your/directory/
```

Similarly, you can copy all the simulation outputs from */champsim/results/* by using the following

```
$ sudo docker cp ucp:/champsim/results/
 ↪  /path/to/your/directory/
```

2) Running the artifact in the background: Tmux (https://github.com/tmux/tmux/wiki), or similar tools, can be used to create a terminal window, launch the job, and detach the terminal. This ensures that the docker image continues to run in the background even if the terminal is closed. To install tmux please check https://github.com/tmux/tmux/wiki/Installing. Once installed you can run the following to create a new tmux session launch the experiments and then detach the session.

```
$ tmux new
$ sudo docker pull
 ↪  sawansingh/ucpisca24:automated
$ sudo docker run -e "RUN_TYPE=parallel"
 ↪  -e "NUM_JOBS=8"
 ↪  sawansingh/ucp_isca24:automated
$ Ctrl + b d
```

To see the list of sessions and then enter the session (session 0 in the example below) please use.

```
$ tmux ls
$ tmux attach -t 0
```

F. Evaluation and Expected Results

Once the artifact has run without error, three tables will appear. The first table is for IPC improvements. The first column show the different variants used, while the second column show IPC improvements (in %). The second and third tables show the H2P accuracy and coverage of *UCP Conf*, respectively.

The manual variant of the artifact also generates an IPC improvement graph in the directory */champsim/pdf*. When the docker is finished, do not close the terminal, before copying the generated graph. To do this, open another terminal and run the following command by replacing the "*/path/to/copy/on/host/*" with the directory you want to save the graph.

```
$ sudo docker cp
 ↪  ucp:/champsim/pdf/fig_ipc.pdf
 ↪  /path/to/copy/on/host/
```

After copying, verify that the PDF was copied correctly, and then the artifact terminal can be closed.

This artifact verifies two main contributions of the paper. First, the improvements in the detection of H2P branches. Second, IPC improvement of UCP (for threshold value of 500). Thus, we have designed the artifact so that after a successful run the following results are available:

- *UCP-Conf*, provides an improved H2P coverage and accuracy compared to TAGE-SC-L (see section VI-A. The Fig 9 shows the graph comparing *UCP-Conf* with the previous state-of-the-art H2P predictor. The improvements are described in detail in section VI-A. *UCP-Conf* provides an accuracy of 14.66% and a coverage of 70%. Accuracy illustrates how many H2P branches mispredict. While coverage indicates how many mispredicted conditional branches are marked as H2P.
- *IPC Improvement*, *UCP* comes with several variants and each of these variants can be verified with the provided artifact. The IPC improvements are *UCP* (main proposal), *UCP-IdealBTBBanking* (*UCP* but with an ideal BTB banking scenario), *UCP-TillL1I* (*UCP* version that prefetches only till L1I and does not decode and fill the µ-op cache), *UCP-SharedDecoders* (*UCP* where no additional decoders are required and the decoders are shared). The following table summarizes the IPC improvement of various *UCP* variants mentioned in the paper.

Variant	IPC Improvement (%)
UCP	2
UCP-TillL1I	1.6
UCP-SharedDecoders	1.8
UCP-IdealBTBBanking	2.2

G. Notes

To run Docker, sudo privileges are required. If the image is being run on a cluster, it is recommended to request the administrator to either grant sudo privileges or create a sudo group specifically for Docker and add the user to that group. For more information, please refer to https://docs.docker.com/engine/install/linux-postinstall/.

H. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html