

JumpStart: Alleviating the Front-End Bottleneck in Modern x86 Processors

LOGAN MOODY, University of Virginia, USA and AheadComputing, USA

BHARGAV REDDY GODALA, AheadComputing, USA

GILLES POKAM*, AMD, USA

ASHISH VENKAT, University of Virginia, USA

With the stagnation of single-threaded performance, the need for optimizations targeting sequential code performance is greater than ever before. Modern workloads have also been growing in code footprint, exposing a bottleneck in the processor front-end, specifically its inability to maintain a sufficiently high fetch throughput to saturate available execution units. To address them, two optimizations exist in modern processors: (1) Fetch-Directed Instruction Prefetching (FDIP) and (2) the micro-op cache (μ op cache), both of which improve fetch throughput leading to improved overall performance. However, since they target the same code regions, in the presence of FDIP, the μ op cache becomes mostly redundant as a performance optimization. This work proposes *JumpStart*: a novel and lightweight mechanism that synergistically exploits hints from FDIP to early-detect a μ op cache miss and wake up the decode pipeline ahead of time, alleviating a key front-end bottleneck.

CCS Concepts: • **Hardware**; • **General and reference** → *Performance*;

Additional Key Words and Phrases: Microarchitecture, CPU, μ op Cache, Recovery, Misprediction

ACM Reference Format:

Logan Moody, Bhargav Reddy Godala, Gilles Pokam, and Ashish Venkat. 2026. JumpStart: Alleviating the Front-End Bottleneck in Modern x86 Processors. *ACM Trans. Arch. Code Optim.* 1, 1 (October 2026), 23 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Modern server workloads are known to have large code footprints and irregular instruction access patterns due to the increasingly complex nature of multi-layer software stacks and high development velocity. For example, a study by Google revealed that the code footprint of *search*, a product under constant development, had been growing at the rate of 27% every year, placing significant pressure on the processor front-end [17]. As processors have continued to grow in size and complexity, the front-end bottleneck has become increasingly apparent, resulting in sharply dropping utilization rates of back-end processor resources.

To tackle this bottleneck, modern processors employ two major optimizations. The first optimization, Fetch-Directed Instruction Prefetching (FDIP) [35], decouples branch prediction from demand

*Work done while at Intel.

New Paper, Not an Extension of a Conference Paper

Authors' Contact Information: Logan Moody, lgm4xn@virginia.edu, University of Virginia, Charlottesville, Virginia, USA and AheadComputing, USA; Bhargav Reddy Godala, bhargav.godala@aheadcomputing.com, AheadComputing, USA; Gilles Pokam, gilles.pokam@amd.com, AMD, USA; Ashish Venkat, ashish@virginia.edu, University of Virginia, Charlottesville, Virginia, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1544-3973/2026/10-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

fetch via a Fetch Target Queue (FTQ), allowing instructions along the predicted control path to be aggressively prefetched into the ICache, ahead of the actual demand fetch. FDIP significantly improves the front-end throughput by avoiding expensive ICache misses and maintaining a steady supply of instructions to the rest of the pipeline. The second optimization, typically employed in processors implementing the x86 instruction set, seeks to eliminate the x86 decode tax that stems from identifying instruction boundaries of variable-length x86 instructions and further breaking them down into internal micro-ops (μ ops), via an in-processor cache called the μ op cache dedicated to storing decoded μ ops from hot code regions. The μ op cache not only reduces the latency incurred by the fetch/decode engine but provides significant power savings by avoiding ICache lookup and bypassing the decoding process, by serving the back-end with cached μ ops when available.

Although these optimizations have been an integral part of modern front-ends over the last decade and have seen an upward trend in transistor investment (i.e., larger FTQs and μ op caches), in this work, we show that this trend is not sustainable in the absence of synergistic policies that boost their effectiveness in a complementary manner. We find that, while FDIP and the μ op cache work well as separate optimizations, they become increasingly redundant from a performance standpoint when combined naïvely, as is done in conventional processor designs. This is because the primary performance benefit of a μ op cache comes from storing decoded μ ops for hot code regions. Incidentally, FDIP also targets the same hot code regions, and as long as FDIP successfully prefetches them into the ICache, the decode pipeline (which is fully pipelined) will be full of instructions and the system will experience little throughput benefit from the μ op cache, particularly on conventional processors with wider decode pipelines (on Lion Cove [24], the decode pipeline can deliver 8 μ ops/cycle whereas the μ op cache can deliver 12 μ ops/cycle, although this is limited by short prediction windows, in particular whether there lies a taken branch within those 12 μ ops.).

Furthermore, Singh, et al. [41] make the observation that scaling the size of the μ op cache yields little performance benefit to a pipeline already featuring FDIP, even when the ICache is not inclusive of the μ op cache. They also identify a key front-end bottleneck that may be attributed to the control path that the processor is resteered to upon misprediction recovery, where misses in the μ op cache and perhaps the ICache could result in long idle cycles. These bottlenecks can be more pronounced on modern applications with large code footprints. To tackle this, they propose a μ op cache prefetching (UCP) mechanism that introduces a second decode pipeline and secondary branch predictor structures to allow for prefetching the alternate path of hard-to-predict branches.

This work proposes a novel low-cost optimization that addresses this critical front-end bottleneck by proactively *jump-starting* the decode pipeline, regardless of whether the bottleneck occurs on the correctly speculated path or the resteered path, by just adding 5-12 bits per μ op cache line, uncovering front-end throughput enhancement opportunities beyond prefetching into the μ op cache along the resteered path. *JumpStart* is a synergistic optimization that leverages hints from FDIP to early-detect a μ op cache miss that occurs farther down the correctly speculated or the resteered path and wake up the decode pipeline ahead of time, enabling a continuous supply of instructions. Specifically, *JumpStart* targets the scenario where a μ op stream spanning multiple cache lines is only partially cached in the μ op cache, requiring a switch to the decode pipeline to demand fetch and decode the remaining instructions. However, since the decode pipeline remains inactive when streaming from the μ op cache [5, 22], this not only entails a switch penalty of one cycle but also runs the risk of experiencing an ICache miss upon demand fetch. By waking up the decode pipeline early, and in particular, while μ ops are still being streamed out of the μ op cache, *JumpStart* can effectively hide long fetch/decode latencies.

However, *jumpstarting* the fetch/decode pipeline is only possible when part of the decoded μ op stream located at the speculated branch target or the branch resteer address (upon a misprediction) already resides in the μ op cache. We find that traditional recency- and hotness-based replacement

policies [36, 48] only focus on improving the hit rate of the μop cache and do not necessarily improve our ability to *jumpstart*. To this end, we also propose a cost-informed μop cache replacement policy that allows for partial streams of critical μops to be prioritized and preserved in the μop cache for longer intervals, enhancing our ability to *jumpstart*, targeting critical scenarios where the execution units are severely underutilized. Our contributions are as follows.

- We uncover a key front-end bottleneck and propose a novel synergistic mechanism called *JumpStart* that wakes up the decode pipeline ahead of time in scenarios where the μop cache only partially holds a decoded μop stream, allowing us to issue early demand fetch requests.
- We show that *JumpStart* provides gains on top of prior works such as UCP [41] and the Misprediction Recovery Cache (MRC) [30], accelerating not only the resteeered path but also the correctly speculated path.
- We perform a significant infrastructure overhaul of the Gem5 microarchitectural simulator (to be released upon publication) to faithfully model modern x86 pipelines by introducing a decoupled front-end with FDIP alongside a μop cache while incorporating critical bug fixes.
- *JumpStart* provides a geomean speedup of 1% on a large collection of SPEC CPU 2017 and server benchmarks, with multiple hard-to-optimize applications (*exchange* at 6% and *text-processing* at 4.5%) gaining significantly.

2 Background

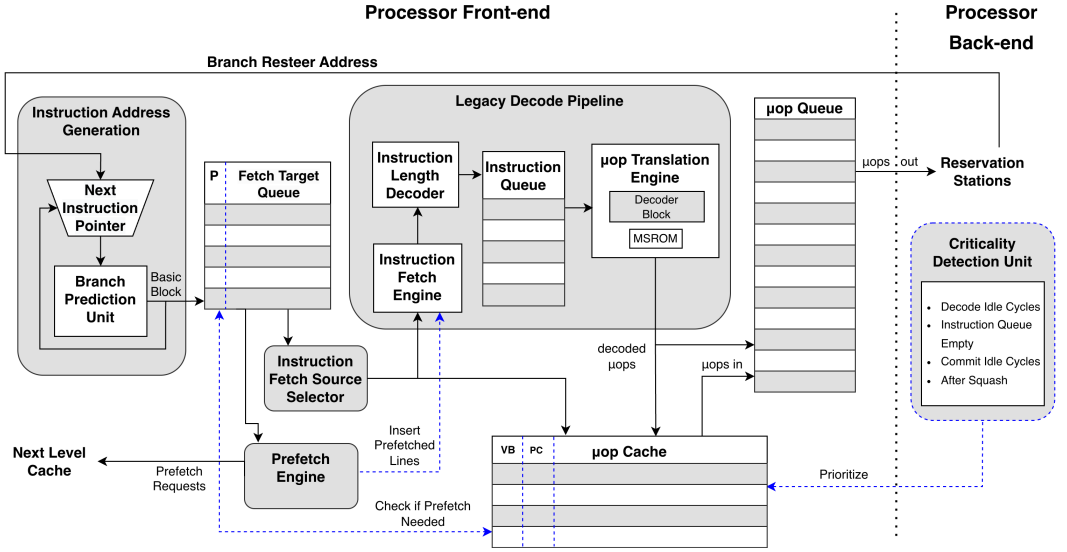


Fig. 1. Overview of the Modern x86 Front-End with our Proposed Extensions (highlighted using blue dotted lines)

The Micro-Op Cache. To mitigate the decode tax, modern processors incorporate a vital component known as the μop cache that stores decoded μops for hot code regions [21, 22, 28, 36, 39, 44, 44–47]. When present in the μop cache, these μops can be swiftly dispatched to the back-end (execution engine) of the processor, bypassing the need for legacy decoders. Compared to the four to five-cycle latency inherent in the legacy decode pipeline [13, 47], streaming instructions from the μop cache incurs a notably reduced latency of one or two cycles. As shown in Figure 1, the instruction fetch source selector steers μops either from legacy decode pipeline or the μop

cache which are placed in a queue of decoded μ ops that feeds into the rest of the processor pipeline. Leveraging the μ op cache enables the processor to fetch and decode a greater number of instructions per cycle than when reliant solely on the legacy decoders. For instance, in Intel's Alder Lake microarchitecture, the μ op cache is capable of streaming up to 8 μ ops per cycle, surpassing the 6 μ op limitation of the legacy decoders [1]. Moreover, when μ ops are being streamed from the μ op cache, the legacy decode pipeline is powered off, resulting in a significant reduction in chip power consumption. However, in the event of a miss in the μ op cache, the fetch engine must stall for one cycle to power on and reset the state of the legacy decode pipeline. The μ op cache is included in the ICache, which means that if a line is evicted out of the ICache, it is also flushed out of the μ op cache.

Fetch-Directed Instruction Prefetching. Modern processors also employ Fetch-Directed Instruction Prefetching (FDIP) [35] that decouples the branch prediction unit and instruction fetch allowing instruction prefetch requests to be issued based on predicted branch targets. As shown in Figure 1, this decoupling is accomplished by adding a Fetch Target Queue (FTQ) between the branch prediction unit and the instruction fetch unit where instead of predicting the next instruction pointer, the predictor predicts the next basic block (start and end addresses) to be fetched and places that prediction in the FTQ [8]. Once entries are placed at the tail of the FTQ, they are broken down into cache lines for which fetch-directed prefetches will be issued. Note that these operations could be pipelined in that the FTQ entry insertion following a branch prediction and the prefetch issue may happen in separate cycles, given the typical multi-cycle latency of TAGE predictors [37]. By the time the FTQ entry makes its way to the head of the queue, at which point, the fetch unit is ready to issue a demand fetch to ICache, the cache line would have potentially already been prefetched into the ICache leading to fewer ICache misses and faster front-end throughput. In essence, the decoupling of the front-end allows for branch prediction and instruction prefetch to run well ahead of the rest of the pipeline allowing cache lines from lower levels of the memory hierarchy to be prefetched into the ICache in time. FDIP remains an active area of research: recent work improves the accuracy and timeliness of fetch-directed prefetches [31] and uncovers branches missing from the BTB within already-fetched cache lines [33]. In fact, FDIP has been shown to hide enough instruction fetch latency that instructions can often bypass the L2 cache [16], an observation similar in spirit to ours regarding the μ op cache.

3 Interplay of FDIP and the Micro-op Cache

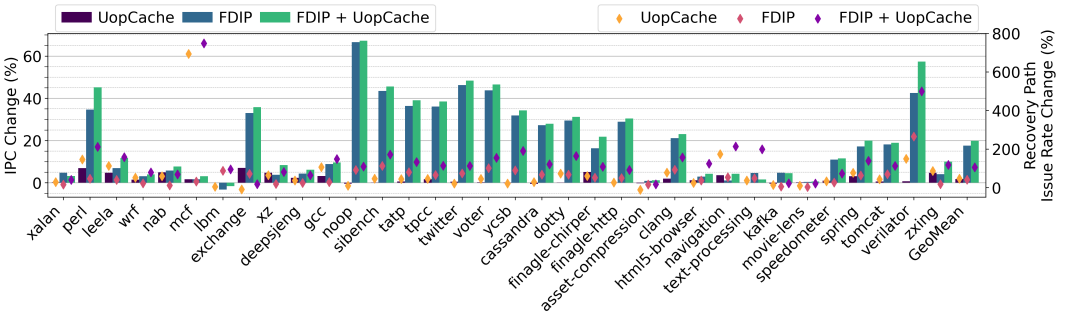


Fig. 2. FDIP and μ op Cache Impact on IPC (bars) and Issue Rate (diamonds)

Like many complex microarchitectural optimizations, FDIP, the μ op cache, and their interactions are notably underdocumented for academic use. To this end, we perform a characterization study

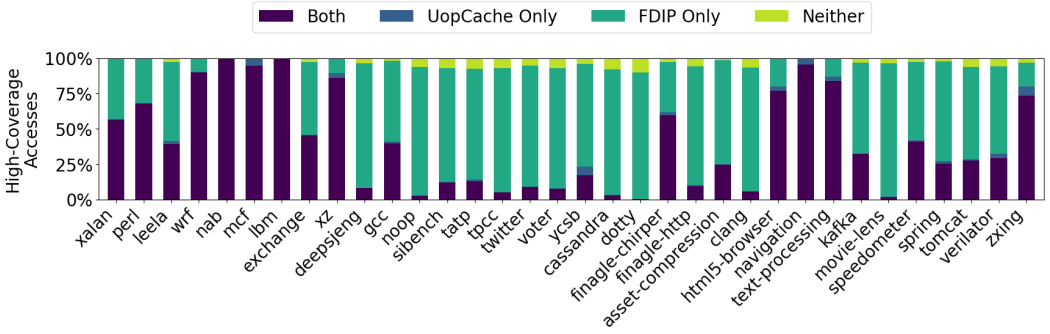


Fig. 3. Breakdown of Instruction Addresses (weighted by frequency) reliably covered by FDIP and the μ op Cache

(methodology in Section 6) to understand their impact in isolation. Figure 2 shows the results of this study. The baseline neither includes a μ op cache nor implements FDIP. We observe that, in a system with only a μ op cache, applications achieve a speedup of 2% on average and as much as 7% on SPEC workloads that are known to exhibit low ICache pressure. On the other hand, in a system with only FDIP present, we observe a speedup of 18% on average and as much as 65% on large code footprint workloads. When combined, many of the workloads show the additive performance of both optimizations. However, workloads such as *nab*, *exchange*, and *gcc* exhibit less performance uplift from the addition of a μ op cache when FDIP is already resident in the system, since the two optimizations target mostly overlapping regions. A small selection of workloads, most notably *verilator* gain significantly more from a μ op cache on top of FDIP. This is because *verilator* experiences a staggering 62 branch mispredictions per thousand instructions (MPKI), due to which the μ op cache’s ability to supply a large number of μ ops immediately following a branch misprediction can drastically affect performance. The right y-axis of Figure 2 highlights this fast recovery by showing the increase in the issue rate in the 20 cycles following a branch misprediction.

Since the gains are mostly additive from both optimizations, we hypothesize that there is a significant overlap in the code regions targeted by FDIP and the μ op cache, due to which the regions covered by FDIP also experience a small boost in performance for being additionally cached in the μ op cache, potentially leaving out code regions that do not benefit from FDIP uncached at both levels. In Figure 3, we classify instruction addresses into four distinct categories. In the absence of a μ op cache, if an instruction is reliably ($>99\%$ of the time) brought into the ICache by FDIP, we attribute it to be *covered* by FDIP (shown in green). Similarly, in the absence of FDIP, if the μ op cache is successful at reliably ($>99\%$ of the time) caching all μ ops of an instruction, we attribute it to be *covered* by the μ op cache (shown in blue). We then examine the intersection (shown in purple and overlapping the two categories above) where the code regions are targeted by both optimizations. The final category shows regions targeted by neither optimization. We observe that FDIP does a remarkable job of supplying instructions in time for demand fetch, and there exists a small subset of instructions that are not well covered by FDIP but do reside in the μ op cache. On a selection of large code footprint workloads, there is a sizeable set of instruction addresses that neither optimization covers reliably, leaving additional headroom to be gained when the two optimizations are combined synergistically.

3.1 Relaxing the Inclusivity Restriction

One of the key consequences of a decoupled front-end that implements FDIP is that the prefetching runs far ahead of the instruction decoding and streaming operations. This implies that aggressive FDIP could potentially evict lines in the ICache and the included μ op cache. Furthermore, since branch predictors make decisions based on prior execution sequences, they will likely predict execution down a hot code region. This results in FDIP issuing prefetches for the hottest regions of code, missing opportunities down the mispredicted path. An inclusive μ op cache would then only ensure that the hottest subset of the lines in the ICache remain resident in the cache. Relaxing the inclusivity restriction allows us to potentially continue to maintain instructions along the resteeered path in the μ op cache, even if aggressive FDIP has evicted the line out of the ICache, thereby enabling faster recovery. While UCP also assumes non-inclusivity [41], there are key differences in how we handle aliases and ICache prefetches.

Aliases. In non-inclusive caches, homonym aliases (the same virtual address from different threads pointing to different physical addresses) may present a problem when the upper-level cache is virtually-indexed and virtually-tagged (VIVT). While we assume a VIVT μ op cache due to its low access latency, we handle homonym aliases by flushing the μ op cache upon a context switch by monitoring changes to control registers such as CR3 that maintains the base address of the top-level page table. Note that the μ op cache is already flushed at enclave boundaries [5, 36]. These overheads are accounted for in our evaluation. Alternatively, we could maintain inclusivity with the iTLB by storing only the iTLB entry index (and certain bits of the virtual tag when the iTLB is partially tagged) in the μ op cache tag array, allowing us to selectively invalidate μ op cache lines upon a TLB shutdown.

On the other hand, synonym aliases may occur when two different virtual addresses may point to the same physical address due to page sharing. Intel processors keep one copy in the μ op cache by enforcing the restriction that no two lines in the μ op cache set may correspond to a single ICache way [5, 41]. This requires sets between the μ op cache and ICache to line up appropriately. AMD processors seem to handle this through physical tagging (also assumed by UCP [41]), which has the drawback of needing a small enough iTLB to meet μ op cache timing requirements. Additionally, an iTLB miss would result in the front-end stalling unless the μ op cache is accessed with a speculative translation, which would then need to be validated later, and require appropriate misprediction recovery handling. Instead, we make the observation that, in the absence of self-modifying code (SMC), aliases in the μ op cache are always consistent. So upon detection of SMC, we flush the entire μ op cache set since synonyms are typically constrained to the same set due to the simple linear mapping scheme used in modern x86 processors [11, 21] (specifically accessed by address bits 5 through 9, which are within the page offset bits).

To ensure correctness during SMC execution, most processors equip the ICache with the ability to receive snoop signals from the L2 cache when a write request is intercepted to an included line. The ICache could then notify the processor core, so that all in-flight instructions up until the offending store get flushed. With a non-inclusive μ op cache, the only change needed would be for the snoop signal to also be forwarded to the μ op cache, which in turn may forward snoop requests to the processor core. This is achievable without introducing new complexity as the ICache already forwards snoop requests to the μ op cache to trigger invalidations [22, 42]. The rest of the mechanisms for SMC handling operate as-is. Our workloads include SMC behavior and we model this effect.

ICache Prefetch Handling. The FTQ needs to remain in full control of the fetch state machine, which means that regardless of the fetch source (i.e., the μ op cache or the legacy decode pipeline), the head of the FTQ will always correspond to the basic block currently being fetched. Thus, in the

presence of a non-inclusive μ op cache, as soon as the branch predictor populates an FTQ entry, before issuing a prefetch to the ICache, the processor must examine the entry to check if the instructions are resident in the μ op cache. If present, the P bit (present in μ op cache bit) of the corresponding FTQ entry is set and the line's hotness is saturated to maximize the chance that it is resident in the μ op cache at the time of demand fetch. Alternatively, we could lock the μ op cache line to prevent it from getting evicted before the demand fetch occurs or store its contents as part of the FTQ entry. However, we go for the former approach due to its simplicity and take a hit in performance in the rare scenario that the P bit is set but the demand fetch misses in the μ op cache (which occurs 0.27% of the time on average in our evaluation). When FDIP processes the FTQ, if the P bit of an entry is set, no prefetch is issued for the corresponding cache lines, thereby effectively utilizing the expanded cache capacity and saving precious prefetch bandwidth. Thus, the μ op cache tag array needs to be looked up once at the time an FTQ entry is populated and once during a demand fetch to stream the μ ops from the μ op cache.

Furthermore, a problem arises when considering that it is possible for a partial hit to occur in the μ op cache. This can occur when probing the μ op cache for a particular cache line, the probe PC is resident but the entire cache line is not. Since μ ops are placed in the μ op cache as they are speculatively decoded it may not even be necessary that entire basic blocks are resident in the μ op cache as FDIP would expect. To this end, we propose to extend the μ op cache tag array to also include the number of bytes stored at the probe PC in the corresponding μ op cache data line (5-bits per line shown as VB (valid-bytes) in Figure 1). This would further allow FDIP to make smarter decisions on what addresses it should issue prefetch requests. Additionally, the μ op cache will require an additional read port to avoid arbitration between prefetch check and fetch unit access. This port, along with the 5 extra bits results in a 1.7% increase in peak power across the core.

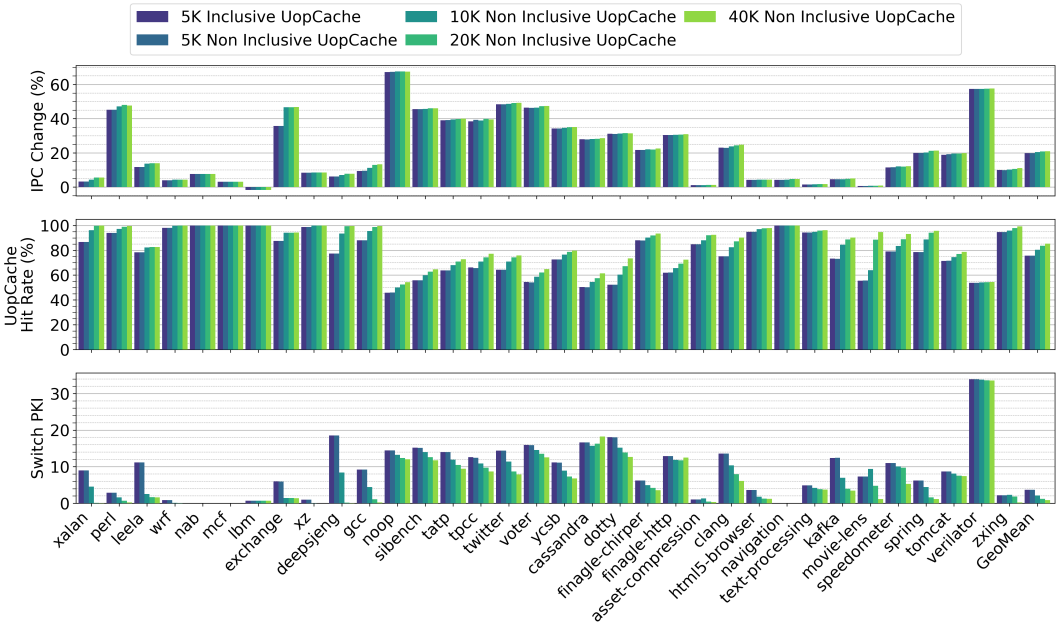


Fig. 4. Impact of Increasing μ op Cache Size on IPC, HitRate (percentage of fetched instructions streamed from the μ op cache), and Switch PKI (switches from μ op cache to legacy decode pipeline per thousand instructions)

3.2 Traditional Scaling Falls Short

We next analyze the impact of relaxing the inclusivity restriction and further examine the effect of scaling the size of the μ op cache as one of the key benefits of non-inclusivity is the effective increase in ICache capacity. Figure 4 shows the results of this experiment. We make the following observations. First, non-inclusivity only provides a small ($<1\%$) speedup on most benchmarks as the μ op cache uses a hotness-based replacement policy and a majority of the lines in the μ op cache continue to be included in the ICache, providing little additional room for other lines along the resteeered path. Second, we observe a small performance benefit with the doubling of the μ op cache size but that starts to taper off at the 20k μ ops limit where the number of code regions cached in the μ op cache exactly matches that of those that can be contained within the ICache. We observe this tapering effect because the μ op cache continues to rely on the ICache for instruction supply and in the absence of an independent supply source, the extra cache capacity added would remain underutilized. Third, since we expand the capacity by increasing associativity, the hit rate of the μ op cache improves with scaling, although that does not necessarily translate into an improvement in overall performance in all applications. In fact, in agreement with our results, the UCP [41] study shows that the mode switching rates vary between applications even if they have the same hit rate, resulting in different overall fetch throughput.

Overall, non-inclusivity alone does not improve performance but enables the preservation of lines in the μ op cache that could get evicted out of the ICache due to aggressive FDIP. The next set of optimizations we describe seeks to identify such lines, showcasing the benefits of non-inclusivity as an enabler of other optimizations.

4 JumpStart: Enabling Faster Recovery

We next propose *JumpStart*, a novel light-weight microarchitectural technique that further improves the synergy between FDIP and the μ op cache, significantly accelerating front-end recovery in x86 processors.

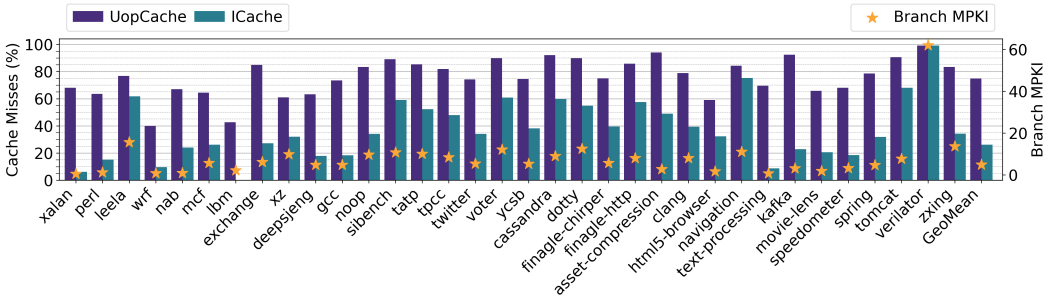


Fig. 5. Percentage of Overall Cache Misses Which Occur Following a Branch Misprediction With Branch MPKI (★)

While FDIP is able to make more aggressive use of the legacy decoders by issuing prefetches, it also incurs a significant penalty following branch mispredictions. In this scenario, not only will the processor incur squash and repair penalties, but the resteeer PC is likely to also result in ICache misses since FDIP only issues prefetches along the predicted path [15]. This means that the execution units of the processor stay fully idle while the cache line is fetched, after which they need to wait even longer for the decode pipeline to fill its internal stages before decoded μ ops can be delivered to the μ op queue, leading to a large number of wasted cycles. We observe this effect

in Figure 5 showing the percentage of total μop cache and ICache misses which occur within a ten-cycle window following a branch misprediction. Since most μop cache misses occur in this small window (a significant portion of which are followed by an ICache miss), we expect *JumpStart* to be most profitable along the resteeered path, although opportunity exists for jump-starting the decode pipeline along the correctly speculated path as well. To fully evaluate its potential along both paths, we propose two variants: (a) *JumpStart* that is only activated upon a branch resteer and subsequent early detection of a μop cache miss, and (b) *JumpStart-Always* (*JS-Always*) that is always active and kick starts the decode pipeline when it early-detects a μop cache miss.

The former scenario is exemplified in Figure 6 which shows the flow of instructions from two different cache lines, A and B, through various stages of the front-end pipeline, following misprediction recovery and branch resteer. In this example, we assume the optimistic scenario where instructions A1 to An that occur along the resteeered path are already present in the μop cache and are thus directly streamed (F(U), i.e., fetched from the μop cache) into the μop queue (Q). However, the subsequent cache line B misses in the μop cache, and as a result, it would need to

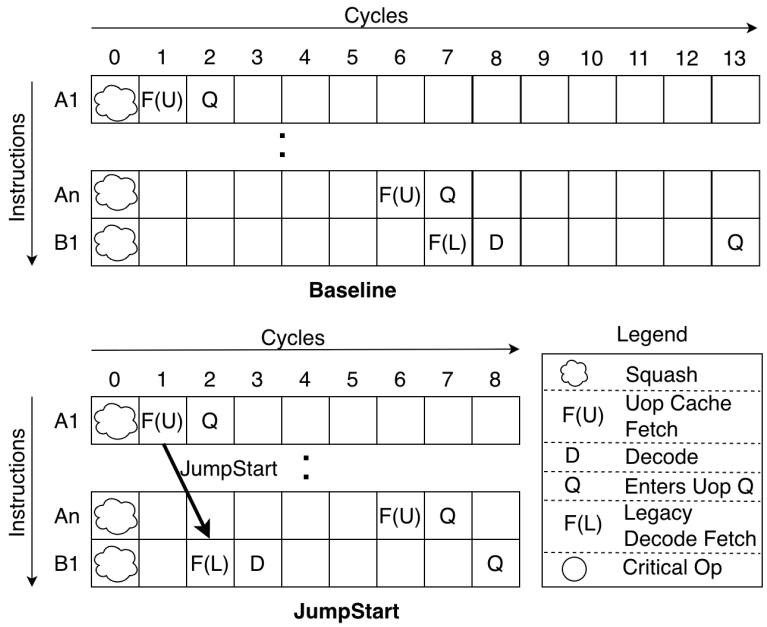


Fig. 6. Timing diagram: *JumpStart* vs. Baseline

be demand fetched at cycle 7 which goes through legacy pipeline (F(L), i.e., fetched via the legacy decode pipeline), before it can be placed in μop queue at cycle 13, and during these cycles the front-end pipeline will remain idle. Note that depending upon how long it takes to service the line from the ICache (and potentially other lower levels, in case of a miss in the ICache, which is likely since FDIP typically does not capture the branch resteeered path), the front-end pipeline could remain idle for an even longer period. However, if we can early-detect that such a miss in the μop cache is expected to happen along the resteeered path and use that signal to wake up the decode pipeline ahead of time, these idle cycles can be substantially mitigated as shown in the bottom half of Figure 6, where front-end *JumpStarts* the decode pipeline in cycle 2, thereby masking the long fetch/decode latency.

JumpStart addresses this issue by synergistically leveraging the capabilities of the μop cache and FDIP. In the first cycle after a branch misprediction, if the resteer address hits in the μop cache, the fetch engine begins to stream directly from the μop cache, deactivating the decode pipeline if enabled. In subsequent cycles, as soon as the head of the FTQ entry is updated by the branch predictor, our early detection logic checks whether the basic block pointed to by the head of the FTQ resides in the μop cache. If not, it activates the decode pipeline ahead of time and instructs it

to start fetching and decoding instructions at that address. *JumpStart-Always* behaves similarly except that the early-detection logic is always enabled, rather than only after branch resteer.

The benefit of *JumpStart* arises specifically from a partial hit along the recovery stream: when the resteer address hits in the μop cache, those μops are streamed to the back-end immediately, and it is this streaming window that *JumpStart* exploits to wake the decode pipeline and issue an early demand fetch for the subsequent line(s) that are expected to miss. As a result, the ICache lookup and the multi-cycle decode latency overlap with μop cache streaming rather than being fully exposed on the critical path. If, instead, the resteer address itself misses in the μop cache, *JumpStart* alone cannot hide the fetch/decode latency. This is exactly the scenario that motivates our criticality-based replacement policy (Section 5), which preserves such critical resteer-path lines in the μop cache, increasing potential *JumpStart* activations.

While the μop cache pipeline continues to deliver a stream of decoded μops until a μop cache miss is encountered, the *jumpstarted* legacy decode pipeline fetches and decodes instructions ahead of time at the address specified by the early detection mechanism. Since the decode pipeline may have μops ready before we have completed streaming from the μop cache, one approach would be to insert a staging buffer between the decode pipeline and the μop queue to ensure a smooth transition when the μop cache is depleted. This buffer may be sized to accommodate enough decoded instructions to cover the latency of the decode pipeline (five cycles in our model). Once the buffer is full, the decode pipeline can be deactivated to conserve power. Another approach is to stall the decode pipeline if the μop cache is still in streaming mode. We take the latter approach since the decoders can deliver 6 μops in the very next cycle that they finish stalling, thereby still ensuring a continuous instruction supply to the back-end. Upon a μop cache miss, the decode pipeline is reactivated if necessary, at which point it begins supplying instructions directly from the staging buffer to the μop queue until it is completely drained, by when, new μops from the re-activated decode pipeline will be ready to fill the buffer.

5 Enhancing the Potential for JumpStart through Criticality-Based Replacement

JumpStart effectively masks the penalty associated with activating and waiting for the legacy decode pipeline to fill, while the μop cache continues to supply the execution units with a steady stream of already decoded μops for immediate execution. However, this is contingent on a partially cached μop stream already available in the μop cache. Figure 7 extends the example in Figure 6 to illustrate this scenario. Specifically, it considers the case when line A misses in the μop cache, forcing a demand fetch from the ICache which could potentially take several cycles (k) to be serviced. Since the pipeline restarts after a squash, any additional latency in delivering instructions to the back-end becomes critical for performance. However, if we are able to deem line A as critical and preserve it in the μop cache for a sufficiently longer period of time (than what a typical hotness-based replacement policy would allow), we would not only be able to avoid the expensive demand fetch and decode, but also fully exploit *JumpStart* for the following line B that is expected to miss in the μop cache, as shown in the bottom half of Figure 7.

State-of-the-art μop cache replacement policies such as FURBYS [48] focus on improving the hit rate by keeping transiently hot code regions in the μop cache. FURBYS is a profile-guided replacement policy that bundles the value (number of μops) and size (number of lines) of a prediction window (PW) stored in the μop cache into a cost metric that gets examined at the time of eviction, in addition to enabling larger PWs to be stored in the μop cache to mitigate partial hits. While this results in a substantial reduction (57.8%) in the μop cache miss rate, it only translates to 0.48% speedup over an LRU baseline as – (1) the latency benefit of the μop cache is more pronounced along the misprediction recovery path which is typically not targeted by hotness-based replacement policies, and (2) small prediction windows hinder its throughput benefit (i.e., when a taken branch

is intercepted before the full fetch bandwidth is reached). Furthermore, to enhance the potential for *JumpStart*, the replacement policy would need to identify critical μop streams that when missed in the μop cache could lead to long idle cycles and at least partially preserve them in the μop cache regardless of their hotness, to provide *JumpStart* with the opportunity to mask the demand fetch/decode latency. Thus, we focus on designing policies and heuristics to enhance the potential for *JumpStart* while offering performance parity with the state-of-the-art replacement policy. Our policy only targets the lines along the recovery path as that is where jumpstarting is most profitable.

Criticality Heuristic.

To identify which instructions cause these long latency recovery scenarios and thus entail the greatest impact on performance, we introduce our definition of criticality. We define a critical instruction as one that is decoded immediately following a squash and for which the decode unit remained idle for a large number of cycles waiting on the instruction. We call the number of instructions the decode unit spent idling as *front-end idle cycles*. During our investigations, we found that for many benchmarks, also considering

back-end idle cycles led to more profitable instructions being discovered, so we require an instruction to achieve a minimum total threshold of front-end + back-end idle cycles. Similar to a front-end idle cycle, a back-end idle cycle is attributed to an instruction whenever a commit is prevented from making progress in a cycle due to this instruction waiting on a memory response at the head of the ROB. We also do not consider instructions following a return as critical, as the target of a return is less consistent than conditional branches since functions are typically called from multiple distinct locations, leading to multiple potential options for prioritization.

To confirm our hypothesis that lines that cause a high number of idle cycles have a greater impact on performance, we perform two experiments. In both experiments, we model a processor equipped with FDIP and an infinitely sized μop cache. In the first experiment, when an instruction is decoded for the first time, its μops are placed in the infinite cache and will remain there until the end of execution, or a snoop request is received for the entry. In the second experiment, we only place a region in the infinite cache when one or more of the instructions in the region meet all of the following criteria for criticality: (1) occurs immediately following a mispredicted branch, (2) causes 10 or more idle cycles at decode where the IQ is also empty (front-end idle cycles), and (3) causes greater than 20 cumulative idle cycles between decode and commit (front-end + back-end idle cycles). Figure 8 shows the results of these experiments. The bottom half of the figure shows the uplift in the Instructions-Per-Cycle (IPC) and the top half shows the number of instructions required

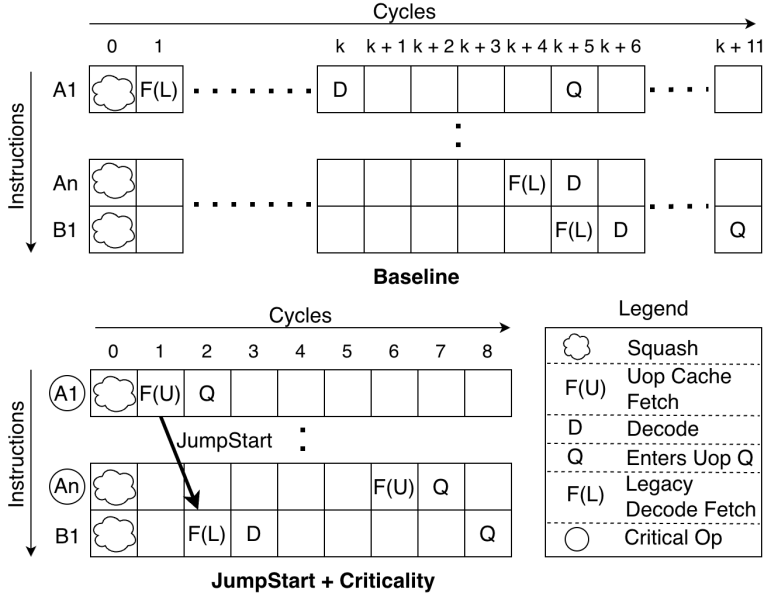


Fig. 7. Timing diagram: *JumpStart* + Criticality vs. Baseline

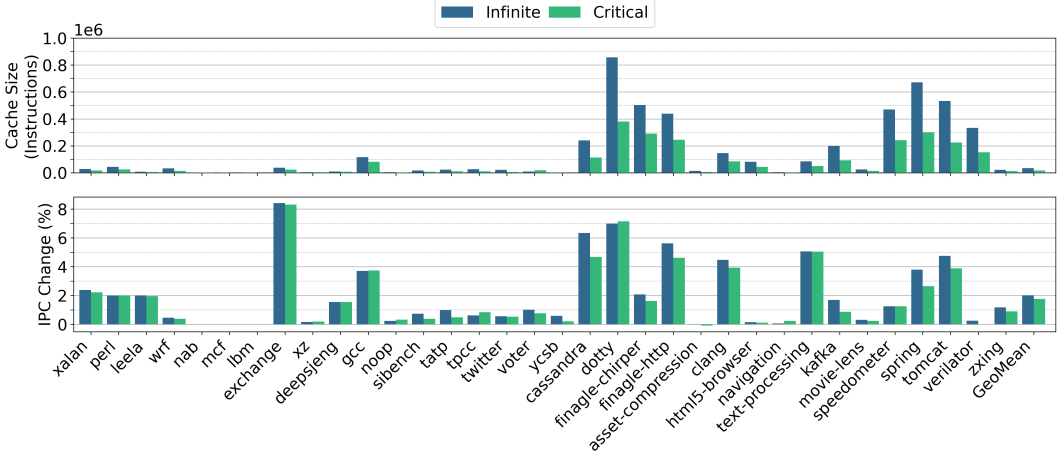


Fig. 8. Speedup: Criticality-Based μ op Cache Limit Study

to be maintained in the infinite cache to achieve that speedup. As shown, the infinite criticality cache is able to reap a substantial chunk of the performance benefits provided by the infinite μ op cache, at a significantly smaller instruction footprint (as much as 8% IPC uplift with close to an order of magnitude smaller storage). However, to provide those gains, the infinite criticality cache still needs to be large (tens of thousands of μ ops), prompting the necessity for a policy that carefully places and preserves critical lines in the μ op cache without imposing additional storage overheads. Since the μ op cache is invalidated upon a context switch to maintain correctness, the maximum performance of both caches is lower than if it were able to maintain state across context switches, however, the performance gain remains nontrivial.

μ op Cache Prioritization and Replacement. We next propose a cost-informed prioritization and replacement policy that selectively places and preserves critical lines in the μ op cache. At commit, we continuously check how many cycles each retiring code region stalls the pipeline, which we refer to as its *cost*. Then, whenever a sufficiently critical region commits (i.e., one with a high *cost*), we forward its *cost* to the μ op cache, which then preserves the corresponding lines for longer than a purely hotness-based policy would.

To calculate this *cost*, we reuse the notions of *front-end* and *back-end idle cycles* defined above and track them using two saturating counters maintained at decode and commit. Recall that a front-end idle cycle is one in which the decode unit has no instruction to process, and a back-end idle cycle is one in which the instruction at the head of the ROB stalls commit while waiting on a memory response. Each counter is 7 bits wide so that it can accumulate up to 127 idle cycles, which is large enough to capture the latency of an LLC miss. Front-end cost is attributed to an instruction by writing the front-end counter value to the ROB entry of the first instruction processed by decode after idling. Back-end cost is attributed to the first instruction that commits after a period of commit inactivity. When an instruction commits and its combined cost exceeds the tuned threshold, we send a prioritization request to the μ op cache carrying (1) the 32-byte aligned uopcache region indexes containing the high cost instruction and a tunable number of subsequent instructions (we refer to this set of regions as the criticality window), and (2) the sum of the front-end and back-end cost.

Upon receiving a prioritization request, the μ op cache looks up each 32-byte index, and upon a hit, attempts to prioritize the entry. To prevent priority lines from dominating any given set of the

cache, we first check whether prioritizing this line would exceed the maximum number of priority lines permitted per set. If the cap is not exceeded, the *priority cost* (PC) bits in the tag array (shown in Figure 1) are set to the cost carried in the request. If the set is already at its priority-line cap, a resident priority line with a lower cost is de-prioritized to make room; if no lower-cost line exists, the request is dropped.

Finally, we observe that some sets are far more likely than others to hold critical lines, thus, we implement a per-set dynamic aging policy for priority costs. Whenever a μop cache line is accessed, we examine the last access of every priority line in its set, and if a line has not been accessed within a tuned half-life (HL) threshold, we halve its stored cost; once a line's cost reaches zero, it is de-prioritized and becomes eligible for eviction by ordinary non-critical lines. Conversely, each time a priority line is accessed, its cost is doubled. Together, these rules let a line's stored cost reflect both how critical the region is and how frequently it is reused.

6 Methodology

Baseline Processor Front-End			
Fetch	8 μops ; stop fetch at taken branch	IDQ	192 entries
ICache	64 KB, 16-way, 1 cycle hit, 24 MSHRs	Branch	TAGE SCL (64KB)
μop cache	5k μops , 8-way, 12 $\mu\text{ops}/\text{cycle}$	I-Branch	ITTAGE (64KB)
Decode	5 cycle latency, 12 $\mu\text{ops}/\text{cycle}$	RAS	36 entries
BTB	8K entries, 8-way	FTQ	24 entries
Prefetch Q	40 cache lines	Fetch Buff	64 bytes
Baseline Processor Back-End			
Dispatch	18 μops	PRF	328 INT/FP
Commit	12 μops	FUs	6-Int, 4-FP, 8-AGU
Repair cost	8 cycles from decode, 15 from execute	LQ/SQ	189/120 entries
DCache	48 KB, 12-way, 3 cycle hit, 24 MSHRs	IQ	318 entries
L2 Cache	2 MB, 8-way, 13 cycle hit, 80 MSHRs	ROB	576 entries
L3 Cache	12 MB, 12-way, 30 cycle hit, 256 MSHRs	Store Set	1024 entries

Table 1. Microarchitectural Configuration Parameters

Baseline Architecture. Table 1 describes our baseline, which is modeled after the Intel Lunar Lake Performance Core (Lion Cove) using the Gem5 architectural simulator. As part of our baseline architecture, we implement support in Gem5 for a decoupled front-end [15] that features both FDIP and the μop cache, in accordance with the Intel Architectures Optimization manual and other prior characterization efforts [5, 8, 13, 35, 36]. Further, since the performance of FDIP depends on the accuracy of the branch predictor, we made several critical improvements. First, we adopted the improved TAGE_SC_L [4] model into our simulator, and enabled speculative updates. Second, we incorporate ITTAGE for indirect branch prediction [38]. Since our baseline features a decoupled front-end, it remains oblivious to potential multi-cycle latency predictions from TAGE, as at steady-state, FDIP, the decode pipeline, and the μop cache will have enough FTQ entries to work with and enable high throughput. Lion Cove is reported to hold 5,250 μops and is able to emit 12 $\mu\text{ops}/\text{cycle}$ and thus 12 $\mu\text{ops}/\text{way}$, with no details on the exact organization. To best approximate this, we implement a μop cache which is 54 sets by 8-ways, where each way can contain 12 μops , which can store a maximum of 5,184 μops . We will make our simulator available upon publication.

Comparison with Related Work. We implement the Misprediction Recovery Cache (MRC) [30] in Gem5 for comparison, albeit adapted to conventional architectures as follows. First, the MRC operates in conjunction with TAGE_SC_L, FDIP, and the μop cache, fetching μops along the

resteered path only upon incurring a μ op cache miss. Second, the bandwidth of the MRC is set to be equal to the bandwidth of the μ op cache, thereby delivering 12 μ ops along the resteered path every cycle. Third, we explore MRCs of varying capacities at which they yield noticeable performance improvements on the workloads we evaluate against. Their sizes range between 1K and 8K μ ops (amounting to 7-56 KB additional storage, assuming an average μ op size of 56 bits from a recent AMD study [22]).

Since UCP [41] is implemented on a different simulator (i.e., Champsim [14]), we aren't able to offer a direct performance comparison. However, we modify the authors' Champsim implementation to identify points at which *JumpStart* could be activated and count the number of idle decode cycles that could be saved due to *JumpStart* on top of UCP. Recall that these scenarios might arise when the μ ops along the resteered path are only partially cached in the μ op cache, which could occur despite prefetching along the alternate path into the μ op cache. We offer a hardware overhead comparison against MRC and UCP in Table 2, the results of the study appear in Section 7.

Benchmarks. We evaluate this work using 35 long-running applications curated from several benchmark suites featuring large code footprint behavior, including SPEC CPU 2017 C/C++ benchmarks (barring some applications that do not work with the latest unmodified version of Gem5 due to lack of support for cer-

tain types of SIMD instructions, privileged operations, and abnormal termination due to memory corruption and/or high memory usage). These are listed in Table 3. Note that these workloads are commonly used to evaluate performance behaviors in datacenters, except for speedometer2.0 and Geekbench. speedometer2.0 is a client workload with complex branching patterns that stress the front-end of the processor, making it a good candidate for this study. In Geekbench, the remaining workloads do not exert pressure on the front-end and are thus not included in this study. Further, a subset of these 35 benchmarks has also been used in prior studies [15, 18–20, 29] that examine the impact of applications with large code footprints.

Technique	Hardware Overhead
MRC [30]	Secondary μ op cache that can host 1K to 8K μ ops
UCP [41]	Secondary decode pipeline with 6 decoders, alternate 24-entry FTQ 32-entry μ op cache MSHR, 32-entry alternate decode queue, 16-entry Alt-RAS alternate 8KB TAGE-SC-L branch predictor (Alt-BP), optional 4KB ITTAGE indirect target predictor (Alt-Ind),
<i>JumpStart</i> & <i>JumpStart-Always</i>	5 bits per μ op cache line, 1 bit per FTQ entry
Criticality-Enhanced <i>JumpStart</i>	12 bits per μ op cache line, 7 bits per ROB entry, one 7-bit saturating counter, 1 bit per FTQ entry

Table 2. Hardware Overhead Comparison

Suite	Benchmarks
SPEC CPU 2017 (C/C++) [43]	xalan, perl, leela, mcf, exchange, deepsjeng, wrf, nab, lbm, xz, gcc
OLTP-Bench (PostgreSQL) [12]	sibench, tatp, tpcc, twitter, voter, ycsb, noop
DaCapo [9]	cassandra, kafka, tomcat
Renaissance [34]	dotty, finagle-chirper, finagle-http, movie-lens
Browser Bench [2]	speedometer2.0
Geekbench [3]	asset-compression, clang, html5-browser, text-processing, navigation
Chippyard [7]	verilator
Spring Dataset & Benchmark [27]	spring
Zebra Crossing Library [6]	zxing

Table 3. Benchmark Description

We compile the benchmarks using -O3 optimization level with LLVM 16.0.6 [25]. We construct representative workloads using the SimPoint [40] methodology as follows. We use PinPlay [32] and Gem5’s atomic CPU mode to generate checkpoints for multiple simulation points that each span 100 million dynamic x86 instructions (6 simpoints per benchmark on average). We use the system call emulation (SE) mode for SPEC CPU 2017 benchmarks (small code footprint) and full system (FS) mode for the rest to measure the impact of system calls, context-switching, and SMC behavior. We warm up for 10 million instructions and then perform a detailed simulation of 100 million instructions using reference inputs.

Unless otherwise noted, performance is reported as the change in IPC over the stated baseline. We compute the geometric mean (GeoMean) over the full benchmark suite for each configuration, and then take the difference from the baseline configuration to get “Change in GeoMean IPC”.

7 Results

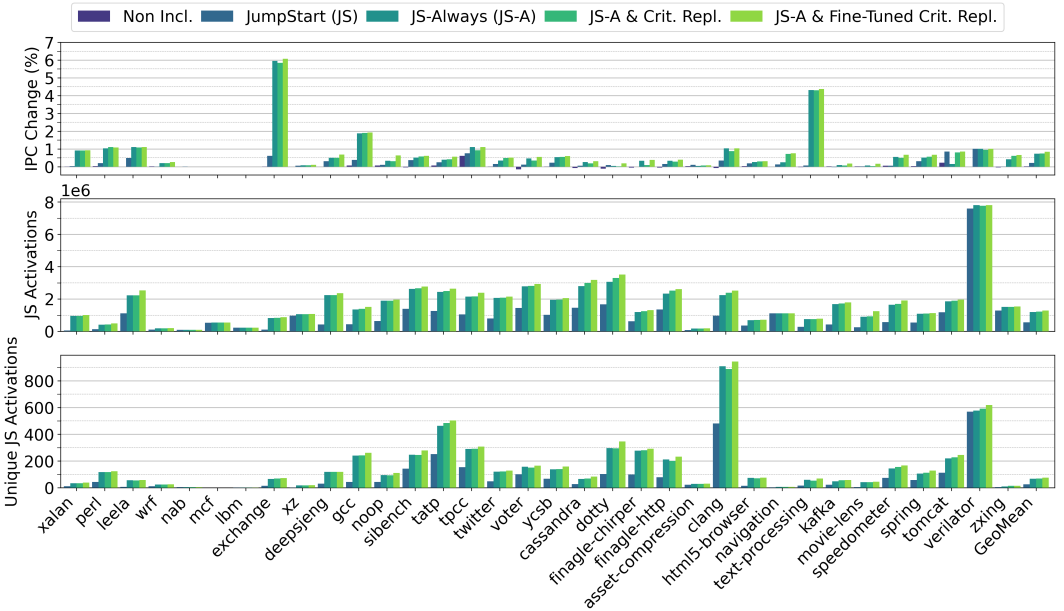


Fig. 9. Performance and Activations Effect of *JumpStart* (JS), *JumpStart-Always* (JS-A), a one-size-fits-all Criticality Replacement Policy and a fine-tuned Criticality Replacement Policy over a baseline with FDIP and a non-inclusive μ op cache.

JumpStart Performance. In Figure 9 we compare the potential of *JumpStart* on the resteeered path as well as the correctly speculated path, against a baseline that features FDIP and an inclusive μ op cache. While the non-inclusivity provides some performance uplift, *JumpStart* provides additional performance gains, particularly in applications with large code footprints, such as those from OLTP-Bench [12], with the number of potential *JumpStart* activations totaling well above a million instances within the 100 million instruction execution interval. The number of activations for *JumpStart-Always* is significantly greater than that of *JumpStart*, ultimately translating to additional gain in IPC. Furthermore, we observe that although the number of *JumpStart* activations hovers over a million instances, the number of unique instructions that trigger a *JumpStart* activation when they hit in the μ op cache is smaller (in the range of hundreds of threads of instructions).

However, on larger code footprint workloads, caching these unique execution threads in a sidecar buffer such as the Misprediction Recovery Cache would be prohibitively expensive, as they could include tens of thousands of μ ops.

Most interestingly, *JumpStart-Always* provides significant additional speedups (as much as 6%) on *exchange*, *gcc*, and *text-processing*. Upon closer inspection, we found that these gains manifest due to a critical feature of our design that improves the synergy between FDIP and the μ op cache. In particular, as previously noted, when a new FTQ entry is generated, a lookup is issued to the μ op cache, and if the result is a hit, a prefetch request is not sent for the corresponding cache lines. This has the benefit of reducing traffic in the cache hierarchy, allowing that bandwidth and capacity to be used for other non-resident lines. However, in some instances, when the FTQ entry is ready to be fetched, the entry may have since been evicted from the μ op cache, in which case the FTQ and prefetch queues must be flushed. We find that it is common that the FTQ entry spans multiple μ op cache lines and that only part of it is usually evicted. Upon a subsequent hit in the μ op cache, *JumpStart* accelerates pipeline recovery.

While it is possible that some of the μ ops speculatively decoded by *JumpStart* could cause pollution in the μ op cache, since these decisions are driven by the FTQ and thus a highly accurate branch predictor, the rate of true pollution remains low. On average, only 0.4% and 0.9% of the overall μ ops placed in the μ op cache during *JumpStart* and *JumpStart-Always* truly result in pollution, meaning that they never commit once placed in the μ op cache. In the common case, even μ ops which do not occur on the correct path when placed in the μ op cache end up on a future correct path before being evicted, resulting in a slight alternate path prefetching effect from *JumpStart*. It is also worth noting that the μ op cache is resilient to pollution owing to hotness-based replacement [36].

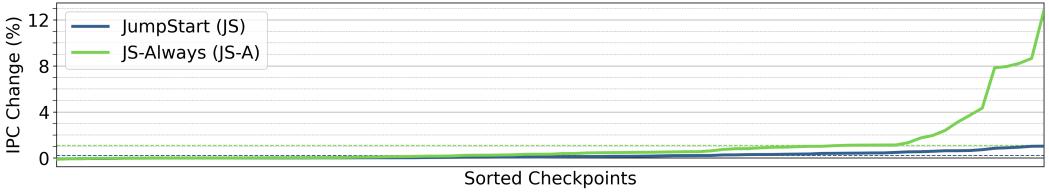


Fig. 10. S-curve of per-checkpoint IPC change for *JumpStart* (JS) and *JumpStart-Always* (JS-A) over the non-inclusive μ op cache baseline. Each point on the x-axis is an individual checkpoint (SimPoint), shown unweighted and uncombined. The dashed lines mark each configuration’s overall GeoMean IPC change.

To provide more detail into the performance gains, Figure 10 presents a sorted S-curve of the per-checkpoint IPC change of *JumpStart* and *JumpStart-Always* over the non-inclusive μ op cache baseline. Unlike the other figures, the x-axis in this figure corresponds to individual checkpoint runs (multiple of which make up a single benchmark result in the other figures). Both variants are positive on the majority of checkpoints, with only a small fraction at the left of the curve showing any degradation, confirming that *JumpStart* rarely hurts performance. *JumpStart* delivers universally modest gains, with most checkpoints clustered near its GeoMean and the best reaching roughly 1%. In contrast, *JumpStart-Always* exhibits a long upper tail: while a large portion of checkpoints see gains in line with its roughly 1% GeoMean, a meaningful number of front-end-bound checkpoints benefit substantially, with the most recovery-heavy regions improving by as much as 12–13%. This tail is driven by the FDIP/ μ op cache synergy described above, in which *JumpStart-Always* accelerates recovery on FTQ entries that were partially evicted from the μ op cache between prefetch and demand fetch. These are the same checkpoints that produce the large per-benchmark gains on *exchange*, *gcc*, and *text-processing* noted earlier. The left portion of the

curve corresponds to the small- and moderate-footprint regions discussed in Section 7, where the instruction working set is largely μop cache resident and a high-capacity MRC can match or exceed *JumpStart-Always*, albeit at considerably higher storage cost.

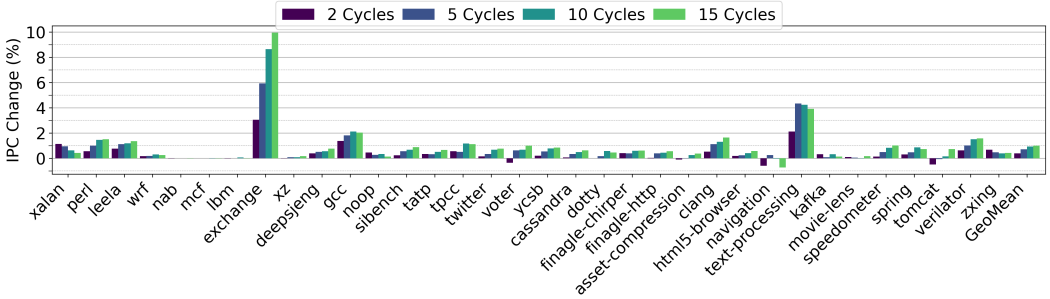


Fig. 11. JumpStart-Always Sensitivity to Legacy Decode Pipeline Delay

When our fine-tuned criticality-based replacement policy is applied over *JumpStart-Always*, we see slight performance gains across all benchmarks, with a 0.2% gain in Geomean IPC, consistent with prior work on μop cache replacement policies [48]. However, we do observe that our criticality-based policy significantly enhances the potential for *JumpStart-Always*, resulting in as many as a million more activations overall as shown in the top graph. Note that our fine-tuned criticality detection mechanism is tuned specifically to each benchmark to show how fine-tuning our replacement policy can remove degradation on benchmarks with many hot code regions, and improve uplift on workloads where additional critical μops are beneficial, essentially mimicking profile-guided heuristics selection [10, 23], again consistent with the recent work on FURBYS, a heuristics-based μop cache replacement policy [48]. For completeness, we also include a one-size-fits-all criticality policy that is not tuned to any specific benchmark. This policy retains some of the performance gains of the fine-tuned policy, but causes degradation on others such as *exchange* and *finagle-chirper*, where the criticality policy is too aggressive and leads to the eviction of useful lines.

As a result of the varying execution characteristics of these applications, some of them benefit from a more or less aggressive replacement policy, as well as targeting a different set of critical lines.

Since one of the key benefits of *JumpStart* comes from masking the decode pipeline delay in addition to potential ICache miss delays (when FDIP isn't able to prefetch well on the recovery path), it is important to examine the sensitivity of *JumpStart*'s performance with varying decode pipeline delays. Figure 11 shows the results of this study. We examine decode latencies of 2, 5, 10, and 15 cycles, with 5 cycles corresponding to the decode pipeline delay in Intel's Lion Cove. We observe that *JumpStart-Always* provides performance gains on pipelines with decode delays that are as small as 2 cycles (potentially mimicking certain RISC pipelines that feature simpler decode pipelines). These gains continue to become increasingly prominent as the delay gets longer (as with architectures exposing complex instruction sets).

To study the impact on power, we model the legacy decoder and μop cache in McPat [26] at the 22 nm technology node. Figure 12 shows energy consumption of *JumpStart*, *JumpStart-Always* and *JumpStart-Always with Critical Replacement Policies*. It shows that only verilator has above a 1% increase in energy consumption with *JumpStart*. The other policies are expected to be higher than *JumpStart*, except for *exchange* and *kafka*, as they show higher performance gains.

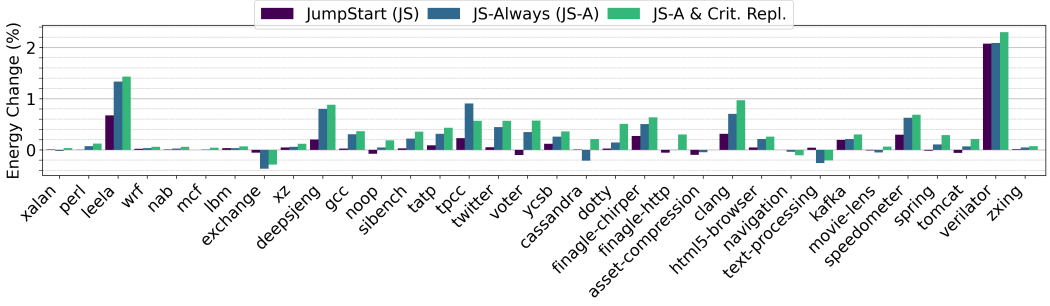
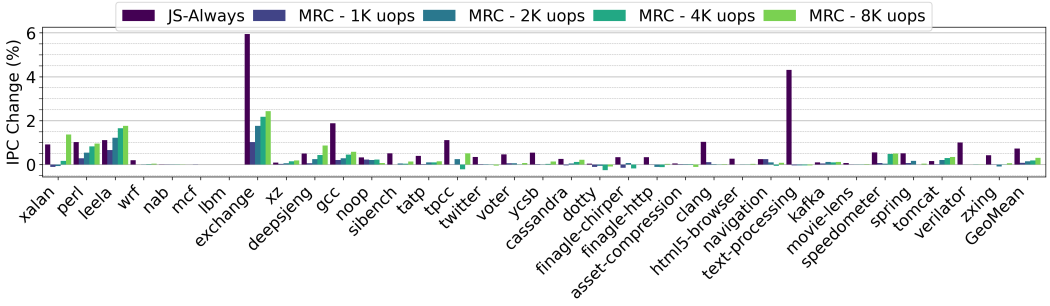
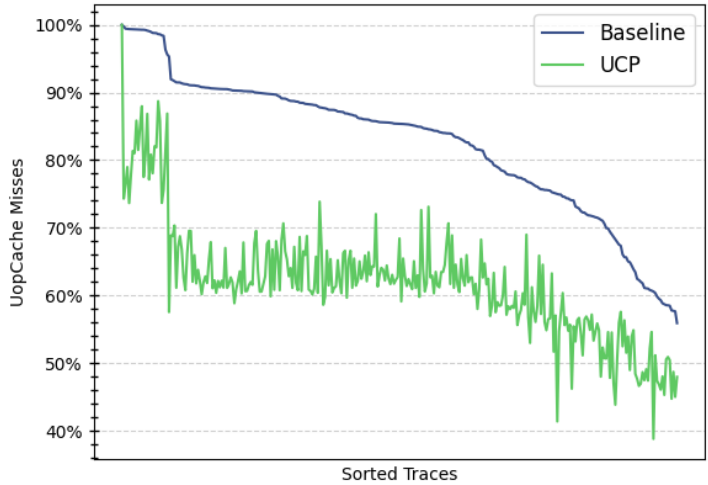
Fig. 12. Impact of *JumpStart* and Critical Replacement Policies on Energy

Fig. 13. Performance Comparison to Misprediction Recovery Cache (MRC)

Comparison with Related Work. In Figure 13, we compare *JumpStart-Always* against MRCs of varying capacities, from 1K μ ops to 8K μ ops. We make several observations. First, on workloads with small code footprints, neither the MRC nor *JumpStart-Always* provides any significant gain. Second, on moderate code footprint workloads such as *xalan*, *perl*, *leela*, and *deepsjeng*, the MRC outperforms *JumpStart-Always*, although it requires considerable storage (roughly 7-56KB) as opposed to *JumpStart-Always* that only adds a few bits per μ op cache tag entry. Third, on applications with large code footprints, such as the OLTP [12], DaCapo [9], and Renaissance [34] benchmarks, even high-capacity MRCs provide little performance gains, while *JumpStart-Always* continues to provide

Fig. 14. μ op Cache misses that occur following a misprediction

steady gains. As shown in Figure 9, the number of branches that mispredict (approximated by the unique *JumpStart* activations) is too large for the MRC to effectively cache.

We next compare against UCP in Figure 14 by reporting the percentage of potential profitable *JumpStart* activations. In particular, we examine the percentage of μop cache misses that occur within 8 cycles of a branch resteer, which Figure 5 shows constitutes the majority of μop cache misses. While UCP achieves its goal of lowering overall μop cache misses, a large number of these misses remain shortly after a misprediction. We note that this analysis uses an 8-cycle window, whereas the characterization in Figure 5 uses a 10-cycle window. The two values correspond to the same window of interest, the number of cycles it takes for a μop fetched from the ICache and decoded by the legacy decode pipeline to reach an execution port. In our *gem5* model this window is 10 cycles but it is only 8 cycles in the UCP ChampSim model [41] used here. We size each window to its respective model so that both capture the misses exposed over this recovery path. In Figure 9 we demonstrate that *JumpStart* can prove an effective means of improving performance within these regions by covering the μop cache miss latency. Thus, the μop cache misses that remain in the presence of UCP are a prime target for profitable *JumpStart* activation. In addition, *JumpStart-Always* could be implemented on top of UCP to hide μop cache miss latency on the correct path.

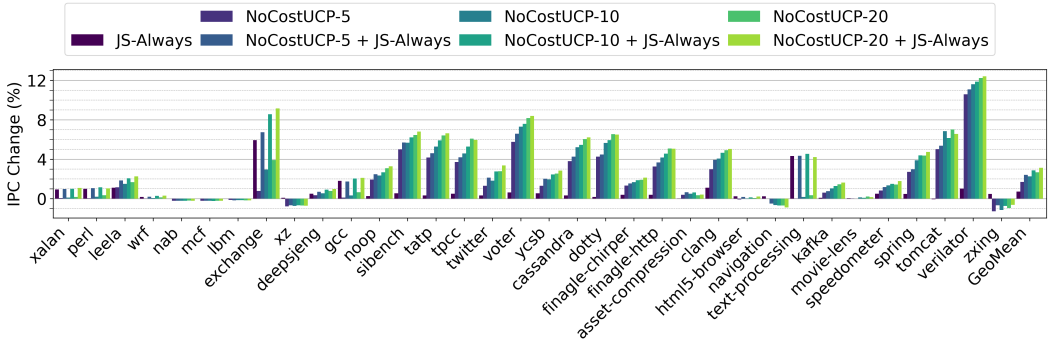


Fig. 15. Performance of *JumpStart-Always* on Top of μop Cache Prefetching (UCP)

To demonstrate the potential benefit over UCP, we compare against a set of optimistic UCP approximations. Our NoCostUCP-*X* experiments are built on top of the same baseline used to evaluate *JumpStart* (FDIP and a non-inclusive μop cache). In addition, we assume a μop cache hit for the first *X* cache lines fetched following a branch misprediction. These lines are not prefetched into the μop cache, thus ignoring the μop cache pollution problem, which, as identified by the UCP authors, causes negative performance effects in a subset of the workloads. This optimistic policy also ignores μop cache placement rules, meaning that it generously assumes a μop cache hit even for lines that would not fit in the μop cache. The UCP authors state that on average, 10 cache lines are prefetched for each alternate path; hence, we model three NoCostUCP policies of 5, 10, and 20 cache lines. Figure 15 shows the results of this evaluation. Across all benchmarks, we continue to observe meaningful performance gains from *JumpStart-Always* over NoCostUCP, even in this optimistic modeling.

Moreover, since UCP only targets μops on the alternate path, the performance gains observed in *xalan*, *perl*, *exchange*, *gcc*, and *text-processing* due to improved recovery from a mismatch between the μop cache contents at prefetch time vs. demand fetch time remain mostly unaffected by any NoCostUCP policy. Our results demonstrate that even up to 20 cache lines after recovery, bubbles

introduced into the pipeline by an eventual μop cache miss can affect performance, and *JumpStart* can remove these bubbles to keep the back-end of the processor fully saturated.

Criticality Exploration. To identify the source of performance gain from our criticality exploration, we incrementally add optimizations to our baseline and identify parameters that achieve maximum performance on the provided set of workloads, by performing a sweep of the parameters shown in Figure 16. We examine how frequently a particular parameter ends up in an optimal configuration for each workload across all possible permutations of the parameter space. First, lines that incur a high number of front-end

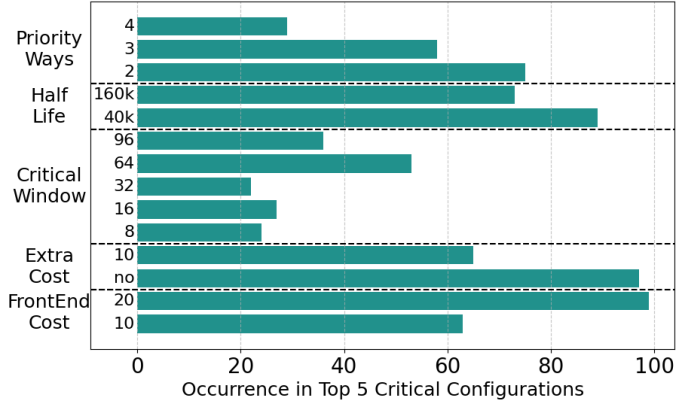


Fig. 16. Criticality Replacement Design Exploration

cycles are preferred, following our claim that instructions that cause front-end stalls create a significant bottleneck in the processor. This is potentially because in the presence of FDIP, ICache misses are less frequent, although the potential for a μop cache miss still exists along the misprediction recovery path. Second, a smaller extra cost (the difference between the combined front-end + back-end cost and the front-end cost) is preferred, meaning that the policy performs better when favoring front-end cycles over back-end cycles. In our other experiments on a smaller core with fewer IQ entries and execution units, we observed this cost to be relatively more important as back-end idle cycles indicate critical events where the whole pipeline gets starved due to a line containing critical long-latency instructions (e.g., a load that misses in the L2 cache) missing in the μop cache. However, on Lion Cove, front-end cycles are significantly more critical as high front-end bandwidth is key to maintaining high utilization of the back-end execution resources. Third, we observe a favored criticality window size of 64 μops , which aligns with the maximum number of μops needed by *JumpStart* to cover the 5-cycle legacy decode pipeline latency (i.e., 12 μops per cycle $\times$ 5 cycles). Recall that smaller criticality window sizes would entail fewer lines being prioritized and larger sizes would likely be rejected as it would require prioritizing too many lines, well above our priority way limit. Fourth, most benchmarks perform best with 2-3 priority ways, since too much contention in the μop cache between priority and non-priority lines can lead to degradation. However, a handful prefer to have a higher maximum number of priority ways with a higher half-life (cycles between halving of the stored priority cost), making the policy more aggressive, allowing critical regions that exhibit concentrated set pressure to be cached effectively with minimal effect on non-priority regions in adjacent sets.

When turned to an optimal strategy, the benchmarks favor criticality thresholds and replacement policies that benefit their unique cache behavior. We observe that *perl*, *leela*, *exchange*, *asset-compression*, *clang*, *html5-browser*, *text-processing*, *speedometer*, and *spring* exhibit lower reuse distances between cache accesses and therefore perform better when the half-life is low and cache turnover is high. Other benchmarks including *cassandra*, *kafka*, *movie-lens*, *verilator*, *xalan*, and *zxing* perform better when critical lines stay in the cache longer due to their longer reuse distances. This analysis also shows that some workloads including *leela*, *deepsjeng*, *asset-compression*, *kafka*, *speedometer*, *xalan*, and *zxing* perform optimally when focusing on any line which experiences a

small number of idle cycles in the front-end. In contrast, many of the other benchmarks benefit from restricting critical lines to those with a higher combined cost.

8 Conclusion

This work examines two key optimizations – FDIP and the μ op cache, and observes that they target similar code regions, resulting in only marginal improvement in performance when they operate in conjunction. To improve their synergy, we propose *JumpStart*, a novel microarchitectural mechanism that activates the decode pipeline ahead of time when the μ op cache only partially holds the μ ops regardless of whether they occur on the correctly speculated or the resteeered path, necessarily requiring the demand fetch of subsequent instructions. Our combined solution can realize up to 2% performance uplift over a baseline implementing both FDIP and the μ op cache, at minimal design and area overheads.

Acknowledgements

The authors would like to thank the anonymous reviewers and Associate Editor James Tuck for their insightful suggestions and comments. This research was supported by NSF grants SSF-2238548, CCP-2213700, and CCP-2217071, and CCI grant HV-2Q26-038.

References

- [1] [n. d.]. AnandTech: Alderlake. <https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3>.
- [2] [n. d.]. Browser Bench. <https://browserbench.org/>.
- [3] [n. d.]. Geekbench cross-platform benchmarks. <https://www.geekbench.com/>.
- [4] [n. d.]. Gem5 TAGE_SC_L Fix. https://github.com/useredsa/spec_tag_scl.
- [5] [n. d.]. *Intel Sandy Bridge Optimization Manual*.
- [6] [n. d.]. ZXing ("Zebra Crossing") barcode scanning library for Java, Android. <https://github.com/zxing/zxing>.
- [7] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (2020), 10–21. doi:10.1109/MM.2020.2996616
- [8] Truls Asheim, Rakesh Kumar, and Boris Grot. 2020. Fetch-Directed Instruction Prefetching Revisited. *arXiv preprint arXiv:2006.13547* (2020).
- [9] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) (OOPSLA '06). Association for Computing Machinery, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488
- [10] William Y Chen, Scott A Mahlke, Nancy J Warter, Sadun Anik, and Wen-Mei W Hwu. 1994. Profile-assisted instruction scheduling. *International Journal of Parallel Programming* 22 (1994), 151–181.
- [11] Shuwen Deng, Bowen Huang, and Jakub Szefer. 2022. Leaky frontends: Security vulnerabilities in processor frontends. In *HPCA*.
- [12] Djellal Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: an extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.* 7, 4 (dec 2013), 277–288. doi:10.14778/2732240.2732246
- [13] Agner Fog. [n. d.]. The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. <https://www.agner.org/optimize/microarchitecture.pdf>
- [14] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR] <https://arxiv.org/abs/2210.14324>
- [15] Bhargav Reddy Godala, Sankara Prasad Ramesh, Gilles A Pokam, Jared Stark, Andre Seznec, Dean Tullsen, and David I August. 2024. PDIP: Priority Directed Instruction Prefetching. In *Proceedings of the 29th ACM International Conference*

- on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24). 846–861.
- [16] Muhammad Hassan, Chang Hyun Park, and David Black-Schaffer. 2025. Second-level Caches: Not for Instructions. *ACM Transactions on Architecture and Code Optimization* 22, 4 (2025), 1–18. doi:10.1145/3769080
 - [17] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 158–169. doi:10.1145/2872887.2750392
 - [18] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *Proceedings of the 54th International Symposium on Microarchitecture (MICRO) (MICRO 2021)*.
 - [19] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO) (MICRO 2020)*. IEEE. doi:10.1109/MICRO50266.2020.00024
 - [20] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA) (ISCA 2021)*.
 - [21] Joonsung Kim, Hamin Jang, Hunjun Lee, Seungho Lee, and Jangwoo Kim. 2021. Uc-check: Characterizing micro-operation caches in x86 processors and implications in security and performance. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 550–564.
 - [22] Jagadish B. Kotra and John Kalamatianos. 2020. Improving the Utilization of Micro-operation Caches in x86 Processors. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 160–172. doi:10.1109/MICRO50266.2020.00025
 - [23] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, Washington, USA) (PACT '06). Association for Computing Machinery, New York, NY, USA, 23–32. doi:10.1145/1152154.1152162
 - [24] Chester Lam. 2025. Lion Cove: Intel's New P-Core. <https://chipsandcheese.com/p/lion-cove-intels-new-p-core>. Chips and Cheese.
 - [25] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.
 - [26] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture* (New York, New York) (MICRO 42). Association for Computing Machinery, New York, NY, USA, 469–480. doi:10.1145/1669112.1669172
 - [27] Lukas Mehl, Jenny Schmalfuss, Azin Jahedi, Yaroslava Nalivayko, and Andrés Bruhn. 2023. Spring: A High-Resolution High-Detail Dataset and Benchmark for Scene Flow, Optical Flow and Stereo. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.
 - [28] Logan Moody, Rasool Sharifi, Layne Berry, Joey Rudek, Jayesh Gaur, Jeff Parkhurst, Sreenivas Subramoney, Kevin Skadron, and Ashish Venkat. [n. d.]. Speculative Code Compaction: Eliminating Dead Code via Speculative Microcode Transformations. *Awaiting Submission* ([n. d.]).
 - [29] Nayana Prasad Nagendra, Bhargav Reddy Godala, Ishita Chaturvedi, Atmn Patel, Svilen Kanev, Tipp Moseley, Jared Stark, Gilles A. Pokam, Simone Campanoni, and David I. August. 2023. EMISSARY: Enhanced Miss Awareness Replacement Policy for L2 Instruction Caching. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) (ISCA '23). Association for Computing Machinery, New York, NY, USA, Article 62, 13 pages. doi:10.1145/3579371.3589097
 - [30] Ashwini K Nanda, James O Bondi, and Simonjit Dutta. 1998. The misprediction recovery cache. *International journal of parallel programming* 26, 4 (1998), 383–415.
 - [31] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. UDP: Utility-Driven Fetch Directed Instruction Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1188–1201. doi:10.1109/ISCA59077.2024.00089
 - [32] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *CGO*.
 - [33] Chrysanthos Pepi, Bhargav Reddy Godala, Krishnam Tibrewala, Gino A. Chacon, Paul V. Gratz, Daniel A. Jiménez, Gilles A. Pokam, and David I. August. 2025. Skia: Exposing Shadow Branches. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. 1091–1106. doi:10.1145/3676641.3716273

- [34] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) (PLDI 2019). Association for Computing Machinery, New York, NY, USA, 31–47. doi:10.1145/3314221.3314637
- [35] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.
- [36] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. 2021. I see dead μ ops: Leaking secrets via intel/amd micro-op caches. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 361–374.
- [37] André Seznec. 2007. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)* 9 (2007), 1–6.
- [38] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2: Championship Branch Prediction*.
- [39] R Sharifi and A Venkat. 2020. CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities. In *ISCA*.
- [40] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review* 36, 5 (Oct. 2002), 45–57. doi:10.1145/635508.605403
- [41] Sawan Singh, Arthur Perais, Alexandra Jimborean, and Alberto Ros. 2024. Alternate Path μ -op Cache Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1230–1245. doi:10.1109/ISCA59077.2024.00092
- [42] Seonghun Son, Daniel Moghimi, and Berk Gulmezoglu. 2025. SMaCk: Efficient Instruction Cache Attacks via Self-Modifying Code Conflicts. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. 1107–1123.
- [43] Standard Performance Evaluation Corporation. [n. d.]. SPEC CPU2017. <https://www.spec.org/cpu2017/>.
- [44] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2018. Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency. In *ISCA*.
- [45] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2022. Mitigating speculative execution attacks via context-sensitive fencing. *IEEE Design & Test* (2022).
- [46] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ASPLOS*.
- [47] Ashish Venkat, Harsha Basavaraj, and Dean M Tullsen. 2019. Composite-ISA cores: enabling multi-ISA heterogeneity using a single ISA. In *HPCA*.
- [48] Kan Zhu, Yilong Zhao, Yufei Gao, Peter Braun, Tanvir Ahmed Khan, Heiner Litz, Baris Kasikci, and Shuwen Deng. 2025. From Optimal to Practical: Efficient Micro-op Cache Replacement Policies for Data Center Applications. In *HPCA*.