Prefetching for Hierarchical Branch Target Buffers

Yongjie Huang
*University of Edinburgh**
Edinburgh, UK
yongjie.huang@ed.ac.uk

Mária Ďuračková
University of Edinburgh
Edinburgh, UK
maria.durackova@ed.ac.uk

Boris Grot
University of Edinburgh
Edinburgh, UK
boris.grot@ed.ac.uk

David Schall
Technical University of Munich†
Munich, Germany
d.schall@imperial.ac.uk

***Abstract*—High-performance CPUs rely on hierarchical branch prediction units (BPUs) to balance the large branch working sets of contemporary workloads against the single-cycle latency required to sustain high fetch throughput. Prior work has overlooked the importance of small, latency-critical L1-BTB and conditional predictor structures, leaving this aspect of microarchitecture unexplored. This paper presents a comprehensive characterization of hierarchical BPUs with an emphasis on L1 structures.**

We find that, relative to an ideal single-cycle BPU, an iso-capacity hierarchical BPU diminishes performance because the latency of L1 misses is often exposed to the fetch pipeline. L1-BTB misses are primarily responsible for performance degradation, even though nearly all branch targets are found in the L2-BTB. Our analysis reveals that L1-BTB misses commonly occur in back-to-back miss chains associated with transitions to new code regions. Branches immediately preceding such chains are often loop exits and function calls/returns, suggesting that inter-region control-flow transitions are a key source of exposed front-end latency. Motivated by this observation, we propose BTB-Ferret, a metadata-free prefetcher for the L1-BTB that exploits existing BTB metadata to avoid storage overheads. BTB-Ferret identifies conditional branches with bidirectional behaviour and function calls, and prefetches BTB entries from the L2-BTB along the alternate path to the predicted one, thereby anticipating upcoming branches. Given a 128-entry L1-BTB, BTB-Ferret covers 37.3% on average (51.9% max) of L1-BTB misses and improves performance by 1.7% on average (9.7% max) while enjoying a simple and low-cost design.

***Index Terms*—BTB, Branch prediction, CPU microarchitecture**

I. Introduction

Today's high-performance CPU cores rely on a decoupled front-end [41], which uses the branch prediction unit (BPU) to identify and predict upcoming branches, placing them into a fetch target queue (FTQ), with the goal of prefetching anticipated instruction blocks into the L1-I cache. The BPU is composed of several structures, including the branch target buffer (BTB) and the conditional branch predictor (CBP). The former tracks branch instructions and their targets, while the latter predicts the direction for conditional branches.

A particular challenge for the BPU is tracking the large branch working set emblematic of contemporary workloads, including in server and mobile domains [6], [57]. For instance, Google's data center workload traces average over 40K unique branches per trace [15]. Tracking such massive branch working sets requires high-capacity BTB and CBP structures. Alas, this requirement is at odds with the need for fast access since at least one branch must be resolved per clock cycle in order to keep the fetch pipeline full. To reconcile the need for large capacity with low access latency, CPU vendors have deployed hierarchical BPUs. Similar to caches, these feature a small-capacity, low-latency first level backed by one or more higher-capacity but slower additional levels. Both BTBs and CBPs employ hierarchical organizations.

For instance, Intel's latest server and client CPUs both feature a 3-level BTB hierarchy [31], [32] while ARM's latest high-performance CPU has a 2-level BTB [33]. The TAGE-based branch predictor, widely used across the industry, does not lend itself to a hierarchical organization [45], [48]. Instead, the first level tends to feature a bimodal predictor (BIM), while the second level uses variants of TAGE-SC-L [51]. If the BIM mispredicts, TAGE corrects a few cycles later.

Existing work has paid considerable attention to accommodating large branch working sets by designing large-capacity last-level BTBs and branch predictors with sophisticated prefetch schemes to deliver branch metadata from the cache hierarchy or a dedicated backing store [11], [27], [45], [48]. However, to the best of our knowledge, none of the works have examined the efficacy of existing small-capacity L1-BTBs and CBPs.

We fill this research gap by conducting a comprehensive characterization of hierarchical BPUs with the focus on the L1 structures. We do this by extending gem5 [36], a full-system simulator that captures wrong-path effects and kernel-level instruction footprint, to faithfully model multi-level BPU hierarchies[1].

Our characterization reveals a high incidence of misses/mispredictions in the L1-BTB and CBP structures, with an average (max) of 29 (76) MPKI for the L1-BTB and 15.1 (60) for BIM. While the vast majority of these are corrected by the L2 structure, the latency of the miss can be exposed to the pipeline depending on the state of the FTQ. Crucially, we find that L1-BTB misses are considerably more important for performance than BIM mispredictions. Compared to an ideal single-cycle large-capacity BTB, a 2-level BTB hierarchy with the same capacity reduces performance by 3.4%. In contrast, a realistic hierarchical BPU incurs only an additional 0.7%

*Work done remotely at Technical University of Munich
†Now at Imperial College London.

[1]https://github.com/yongjiehuang/BTB-Ferret

performance loss compared to an idealized single-cycle BTB + BIM + TAGE-SC-L complex.

A key reason why L1-BTB misses are so critical is that they tend to correlate with transitions to new regions of code, whose code footprint might also not be in the L1-I. While instruction prefetching via the FTQ can help, L1-BTB misses greatly diminish prefetch timeliness, thus hurting front-end performance. Moreover, we find that L1-BTB misses tend to occur in back-to-back *miss chains*, which also correlates with transitions to new code regions and contributes to diminished efficacy of instruction prefetching from the FTQ. A deep dive into BTB accesses that immediately precede miss chains shows that they are commonly branches corresponding to a loop exit and function returns, lending further credence to the notion that region transitions are responsible for much of the performance loss in hierarchical BTB organizations.

Based on the findings above, we identify prefetching into the L1-BTB as a promising direction. However, we find that stateful prefetchers, such as Markov [56] and temporal streaming [16], [25], necessitate exorbitant storage overheads of tens of KBs owing to the need to track individual branch PCs. With many of our studied workloads having active BTB working sets in excess of 10K branches, the storage requirements for a stateful prefetch dwarf the L1-BTB, commonly just a few KBs in size in order to allow a single-cycle access in a high-frequency core.

Given that restriction, we ask whether a perfetcher without any metadata is possible. We identify timeliness as a key challenge – prefetching in the critical path of a miss chain (i.e., once the first L1-BTB miss has been encountered) is clearly not helpful. Meanwhile, anticipating L1-BTB misses is impossible in a metadata-free (stateless) design. Our key insight is that it is the "alternate" path of a branch that tends to trigger future BTB misses. For instance, for a loop, it is the loop exit direction; for a function call, it is the code after the return.

Based on this simple insight, we propose BTB-Ferret, a metadata-free prefetcher into the L1-BTB. Using only existing BTB metadata, BTB-Ferret triggers N-deep prefetching upon encountering a conditional branch with alternating behavior or a function call. Prefetches are generated using simple logic and are placed into a small-capacity prefetch buffer to avoid thrashing the L1-BTB. Despite its simplicity, we find that BTB-Ferret is highly effective. For a 128-entry L1-BTB and a 32-entry prefetch buffer, BTB-Ferret covers 37.3% of all L1-BTB misses, improving performance by 1.7% on average (9.7% max).

Our contributions can be summarized as follows:

- Hierarchical BTB organizations introduce performance overheads due to capacity constraints in the L1 structures owing to the need for single-cycle access. Compared to an ideal single-cycle BPU, an iso-capacity hierarchical BPU experiences a 4.1% average performance degradation.
- Within a hierarchical BPU, the BTB is the acute pain point, partly due to its importance to L1-I prefetching. We find that L1-BTB misses often occur in chains, triggered by transitions to new code regions and preceded by branches that initiate such transactions (e.g., loop exits, function calls/returns).
- The active BTB branch working set often exceeds 10K branches, dwarfing the capacity of a typical L1-BTB. The fine-grained nature of branches that the BTB tracks and the need to track many branches make a storage-intensive stateful prefetcher a poor match for the L1-BTB.
- We propose BTB-Ferret, a metadata-free prefetcher for the L1-BTB. BTB-Ferret is powered by a new insight that only certain types of branches precede the bulk of L1-BTB misses. BTB-Ferret uses existing BTB metadata to identify such branches and triggers prefetching on the *alternate* path of the predicted one. In doing so, BTB-Ferret anticipates upcoming code region transitions and prefetches the upcoming L1-BTB working set. With a 128-entry L1-BTB, BTB-Ferret covers an average of 38.8% of all misses and improves performance by 1.7% while enjoying extreme microarchitectural simplicity.

II. Background and Motivation

A. Core Front-end Basics

Today's high-performance CPUs tend to implement a decoupled front-end [41], whereby the branch prediction unit (BPU) continuously generates a stream of fetch targets that are placed into a Fetch Target Queue (FTQ). The Instruction Fetch Unit (IFU) consumes targets from the head of the FTQ, while the targets queued behind the head are used to issue prefetches for upcoming instruction blocks. Fig. 1 shows a canonical decoupled front-end architecture.

The decoupled front-end architecture is attractive in that it effectively provides an instruction prefetcher at negligible storage and complexity cost. Alas, the approach fundamentally relies on a highly accurate BPU, whose chief elements are a branch target buffer (BTB) and the conditional branch predictor (CBP). The BTB stores the target for each tracked branch, along with the branch type (e.g., call, conditional branch). The CBP relies on the BTB to identify branches so that a direction prediction can be made. A miss in the BTB means that an upcoming branch is overlooked, resulting in an incorrect (pre)fetch stream if the branch is taken.

A BTB miss or a branch misprediction can take tens of cycles to resolve, resulting in degraded performance due to front-end stall cycles, thrashing in the L1-I due to wrong-path (pre)fetches, and degraded energy efficiency due to wrong-path execution. In order to minimize the BPU MPKI, it is essential to use sufficiently large BTB and CBP structures to accommodate the application's branch working set. Alas, characterizations of server and mobile workloads point to vast front-end working sets of tens of thousands of branches [5], [6], [24], [55], [57] necessitating massive BTB and CBP capacities to accommodate them.

In order to avoid fetch bubbles in the face of large instruction (and, correspondingly, branch) working sets, today's

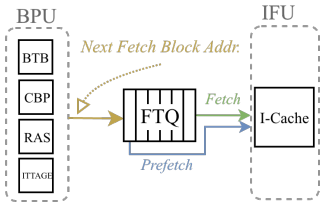


Fig. 1: Decoupled Front-End

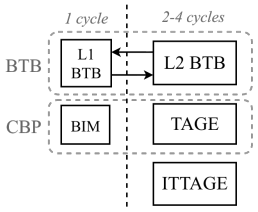


Fig. 2: Latency Hierarchy

high-performance CPUs have deployed large-capacity BTB and CBP structures.

Consider the BTB, which, for each tracked branch, must store the full target address along with the tag, branch type, and potentially additional metadata bits, resulting in a per-entry storage requirement of around 8B [5] A 16K-entry BTB, which is a representative BTB capacity in today's CPUs, including Arm Cortex X925 [33] or AMD Zen5 [4], would thus necessitate 128KB of capacity. While techniques for compressing target fields [5] and fusing multiple branches into one entry [4] exist, they do not eliminate the need for large BTB storage budgets.

On the CBP front, today's processors rely on a multi-component predictor complex, the core of which is the TAGE branch predictor [50]. TAGE relies on a multitude of tables, each indexed with a different length of global branch history; the table with the longest history for the branch is used to provide the prediction. A single hard-to-predict branch may have thousands of TAGE entries, each entry requiring approximately 12 to 16 bits of storage [48]. As a result, today's CPUs commonly feature 64KB or larger TAGE-based CBPs [2], [30], [51].

The downside of large physical arrays, such as those needed for the BTB and TAGE, is that they require multiple cycles of latency to access. For instance, benchmarking has revealed that the ARM Cortex X925's 16K-entry BTB has an access latency of up to 3 cycles [33]. Clearly, multi-cycle access latencies to BPU components are at odds with the ability to generate a prediction every cycle. While ahead pipelining [52] could help hide BTB access latency, it would still be exposed upon a front-end resteer. Moreover, the utility of ahead pipelining for BTBs is unknown.

B. Hierarchical BPU Organizations

In order to reconcile the need for both high capacity and low latency, high-performance CPUs have resorted to deploying hierarchical BPU organizations, an example of which is shown in Fig. 2. Similar to caches, the idea is to provide a small and fast first level, backed by larger but slower additional level(s).

On the BTB front, x86 and ARM CPU vendors employ hierarchical organizations with two or three levels. For instance, Intel's Redwood Cove core (used in its latest Granite Rapids server CPUs) features a 3-level BTB hierarchy composed of 128-, 6K- and 12K-entry BTBs for a combined BTB capacity of ∼18K entries. The smallest BTB has a single cycle latency, while the largest has a latency of 3-4 cycles [32]. Arm's Cortex X925, its latest high-performance core, features a 2-level BTB

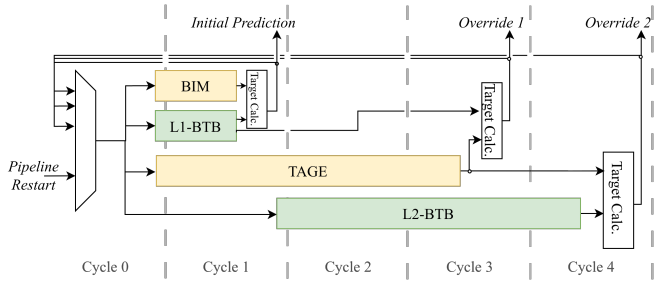


Fig. 3: BPU overriding scheme.

hierarchy. The first level can track up to 2K entries (actual capacity depends on branch spacing), with a reported single-cycle access latency. The second level stores up to 16K entries with an access latency of 2-3 cycles [33].

For the CBP, the tiering strategy is different than that for the BTB because TAGE does not directly lend itself to a hierarchical organization [48]. Instead, the small-and-fast first level of the CBP is served by an untagged bimodal (BIM) predictor, commonly with 4-8K entries (1-2KB of storage). The tagged tables of TAGE, being much larger in size, require 2-3 cycles to access. Since a prediction is required on each cycle to avoid fetch bubbles, the BIM supplies it and TAGE subsequently corrects it, if need be (this is called an *override*). Crucially, the FTQ can hide the override provided that the corrected entry has not yet made it to the head of the FTQ.

C. Hierarchical BPUs: No Free Lunch

While intuitively attractive, hierarchical BPU organizations do not come for free when it comes to performance. For instance, a miss in a higher-level (smaller capacity) BTB results in a serialized access to the next BTB level, causing BTB-induced stall cycle(s). An override of a BIM prediction by TAGE forces a resteer and multiple front-end bubbles.

We quantify the effect of a hierarchical BPU on performance by comparing a realistic 2-level BPU (*baseline*) to an idealized single-level BPU with the same capacity as the hierarchical design but an unrealistically low, single-cycle access latency (*Ideal BPU*).

For the 2-level baseline, we model a block-based BTB organization with a 128-entry, single-cycle L1-BTB and a 16K-entry L2-BTB with a 3-cycle access latency. Note that the L2-BTB is accessed only if the L1-BTB misses; hence, the total latency for an L2-BTB hit is 4 cycles.

For the CBP, we model TAGE-SC-L [51] with overriding. The 8K-entry (2KB) BIM provides its prediction within one cycle, while TAGE and the statistical corrector (SC) ensemble override BIM's prediction, if necessary, 2 cycles later. Note that TAGE-SC is always consulted for a conditional branch; hence, it is looked up in parallel with the BIM, thus hiding one cycle of its 3-cycle latency. Fig. 3 shows the timings of the various hierarchical BPU components in the context of a pipeline.

We model ITTAGE [49] for indirect branches and calls with changing targets. If ITTAGE provides a prediction for such a branch, we model a 3-cycle prediction delay. For an indirect

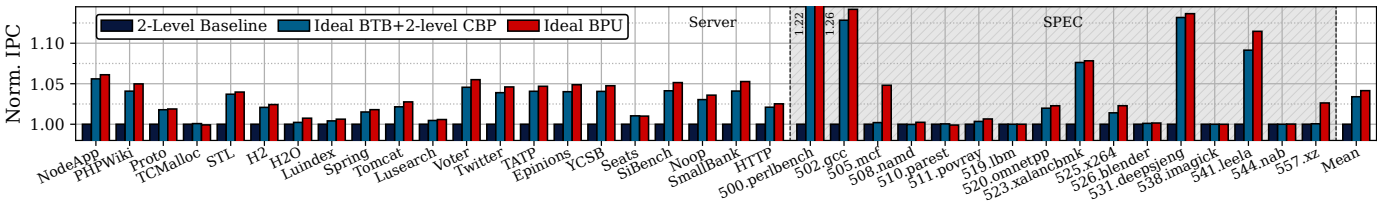
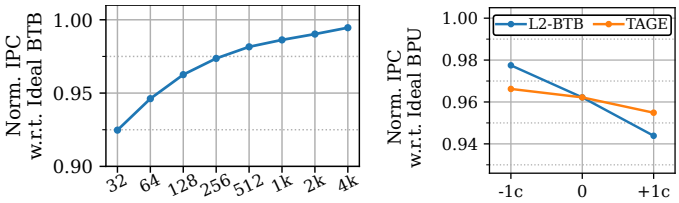


Fig. 4: Performance of a realistic 2-level BPU (baseline) against idealized variants. *Ideal BPU* has the same capacity as the baseline but single-cycle access latency to BTB and CBP. *Ideal BTB + 2-level CBP* idealizes only the BTB component.



(a) L1-BTB size (b) BPU latency

Fig. 5: Sensitivity to L1-BTB size and BPU latency. +/-1c: plus/minus one cycle of latency for the L2-BTB and TAGE.

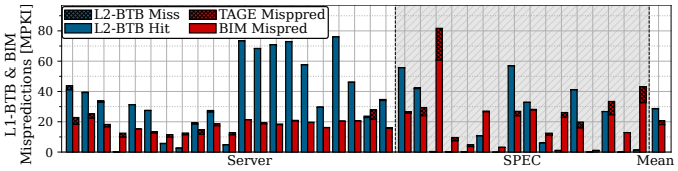


Fig. 6: L1-BTB misses and BIM mispredictions.

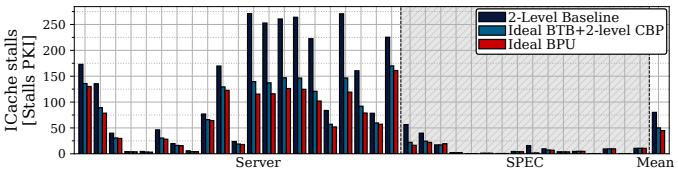


Fig. 7: Stall cycles due to L1-I misses.

branch with a constant target, the BTB provides the prediction, which determines its latency.

We simulate everything in gem5 [18], [36], thus capturing wrong-path effects at the microarchitecture level and kernel-level footprint at the code level. Our implementation faithfully models false-path fetching prior to an override. Specifically, when the L1-BTB misses or the BIM mispredicts, we continue pushing fetch targets into the FTQ, potentially resulting in false-path prefetches to be issued until the lower level of the BPU hierarchy overrides. For the sake of implementation simplicity, override fetch targets are only used to issue false-path prefetches and are not consumed by the fetch unit, avoiding the complexity of early resteers or partial flushes. Section VI provides further details on the simulator, microarchitectural parameters and the workloads.

Fig. 4 shows that an ideal BPU delivers a 4.1% average (25.5% max) performance improvement over the realistic 2-level baseline[2]. The performance difference is strictly due to the latency overhead of the 2-level BPU baseline, since the capacity of the ideal and baseline BPUs is the same.

To understand which BPU component is chiefly responsible for the difference in performance, we perform an ablation study in which we idealise only the BTB component while modelling a realistic CBP (single-cycle BIM, 3-cycle TAGE). The result is shown by the middle bar of Fig. 4. The figure shows that idealising just the BTB provides a 3.4% average (21.6% max) performance improvement over the 2-level baseline, capturing nearly all of the benefit provided by an ideal single-cycle BPU. Reducing, in addition, the latency of TAGE to a single cycle improves the baseline's performance by just 0.7% (to 4.1% in total), on average.

[2]All variants have a 3-cycle access latency to ITTAGE. We found that reducing ITTAGE latency to a single cycle has a negligible impact on performance.

Fig. 5a shows the sensitivity to the size of the L1-BTB. As expected, a larger L1-BTB mitigates the performance impact of a hierarchical organization. However, even at 1K entries in the L1-BTB, a 1.4% performance gap remains between a hierarchical BTB and the Ideal.

Fig. 5b examines the performance sensitivity to the latency of the L2-BTB and TAGE. Consistent with the previous findings, we find a high correlation between L2-BTB latency and IPC. Increasing the L2-BTB access latency from 3 cycles to 4 cycles diminishes the IPC by 1.8%. However, even with a 2-cycle L2-BTB latency, the IPC is 2.2% lower than that of an ideal single-cycle BTB. In contrast, changing the latency of TAGE has a rather modest impact on performance, with IPC varying by just +/-0.5% with a 1-cycle faster/slower TAGE access latency.

D. Why a BTB Hierarchy Hurts Performance

We now tackle the question of why the core is so sensitive to the access latency of the L2-BTB, particularly when compared to that of TAGE[3]. We start by analyzing L1-BTB misses and BIM mispredictions.

Fig. 6 shows the L1-BTB and BIM MPKI as a function of whether the L1-BTB miss or BIM misprediction is satisfied by the next level of the BPU or not. We draw several conclusions from the data. Firstly, we note that the miss rate in the first level of the BPU is fairly high, averaging 29 MPKI for the L1-BTB (max 76 for SiBench) and 15.1 MPKI for the BIM (max 60 for 505.mcf). Secondly, we note that the miss rate for the L1-BTB is generally higher than for the BIM, with the exception of just a few workloads. In fact, seven of the

[3]Unless stated otherwise, we use the term TAGE to refer to the entire TAGE-SC-L complex excluding the BIM

studied workloads have an L1-BTB MPKI exceeding 50 (!), whereas just one workload has a similarly high MPKI for the BIM. Third, we note that nearly all of the L1-BTB misses are satisfied by the L2-BTB; in fact, only a few workloads have a non-negligible L2-BTB miss rate. For the BIM, we observe that TAGE corrects the majority but not all of the mispredictions.

The findings above partly explain the much-higher sensitivity of the core to L1-BTB misses as compared to BIM mispredictions. Not only are L1-BTB misses more frequent than BIM mispredictions, they also take longer to resolve by the L2-BTB (3 cycles) as compared to TAGE (2 cycles). However, that does not fully explain the performance sensitivity observed in Section II-C, which showed that an ideal BTB provides a 3.4% IPC improvement over a 2-level baseline, whereas an ideal CBP yields only 0.7% better IPC.

Our investigation reveals that the BTB's oversized role in instruction cache prefetching is the other major reason for the performance sensitivity to the L2-BTB's access latency. Prior work on front-end prefetching has established that a lot of instruction cache misses occurs when the control flow transitions to a new code region, which commonly happens as a result of a function call or unconditional jump [27]. It is the role of the BTB to identify the corresponding control transfer instructions that do not require direction prediction by the CBP. Meanwhile, conditional branches tend to have short offsets, typically spanning just a few cache blocks, thus contributing little to instruction prefetch coverage [5], [28]. Thus, misses in the L1-BTB for branches that cause a transition to a new code region are particularly hurtful since they preclude timely prefetching of instructions and (as we show Section III-B) tend to trigger a sequence of back-to-back misses in the L1-BTB for branches in the new region of code.

Fig. 7 substantiates the discussion above by showing stall cycles per kilo-instruction (CPKI) caused by instruction cache misses. The studied configurations are the same as in Fig. 4, with the 2-level BPU as the baseline, Ideal BTB having a single-cycle access latency to the full 2-level BTB capacity, and Ideal BPU having a single-cycle latency to all components of the BTB and CBP.

As the figure shows, I-cache misses are responsible for a significant number of stall cycles, with an average stall CPKI exceeding 75. An ideal BTB provides a substantial reduction of 38%, on average, in I-cache stall cycles. Meanwhile, Ideal BPU (which improves on the Ideal BTB by also making the entire CBP single cycle), provides only a small additional reduction of 11%, on average. This study allows to conclude that, in a realistic 2-level BPU, L1-BTB misses diminish the decoupled front-end's ability to effectively prefetch instructions, exposing the core to an elevated incidence of I-cache stall cycles.

The figure also explains the seemingly surprising result in Fig. 4, where SPEC workloads are more sensitive to L1-BTB misses than server workloads. The result may appear counterintuitive because server workloads are known to be

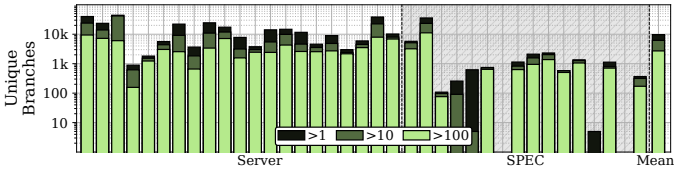


Fig. 8: Unique branches that miss more than 1, 10, and 100 times in the L1-BTB. The missing bars correspond to 519.lbm and 544.nab, which have no L1-BTB misses.

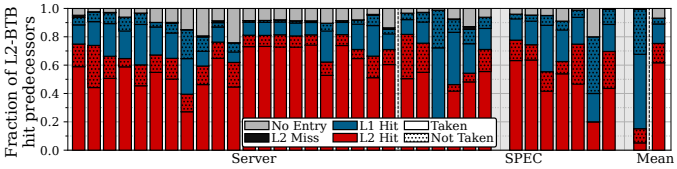


Fig. 9: Breakdown of L2-BTB hits by predecessors

front-end bound, whereas SPEC workloads are not. The reason why SPEC workloads do, in fact, show significant benefit from an ideal BTB is because their instruction footprint tends to fit in the L1-I but not in the L1-BTB. In the absence of I-cache misses (per Fig. 7), the fetch unit can sustain high instruction delivery throughput to the core, provided the BPU-driven address-generation logic can produce fetch targets fast enough. However, L1-BTB misses introduce bubbles in the address-generation pipeline, degrading fetch efficiency on SPEC workloads in the presence of a hierarchical BTB.

III. Understanding L1-BTB Misses

Having identified L1-BTB misses as a key bottleneck limiting front-end performance, we conduct a root cause analysis to understand and identify opportunities to overcome them.

A. L1-BTB Working Set

As revealed in Fig. 6, across most benchmarks, the L1-BTB miss rate is remarkably high, with an average of 29 MPKI and reaching 76 MPKI for SiBench. At the same time, the vast majority of L1-BTB misses are absorbed by the L2-BTB: only 0.86% of L1-BTB misses also miss in the L2-BTB.

To quantify the size of the active branch working set, for each static branch, we track how often it misses in the L1-BTB. Fig. 8 plots the number of unique branches that miss more than once, ten, and a hundred times over the 200M instruction simulation window. We find that, on average, 9.6 K branches miss more than once, reaching up to 44 K for one benchmark (Proto). Of these, 6.2 K miss more than ten times, and 2.7 K more than a thousand times per simulation window[4]

Take-away: the L1-BTB is overwhelmed by a large branch working set that is orders of magnitude larger than what it can hold. Misses for a given branch tend to recur many times.

B. L1-BTB Misses: When It Rains, It Pours

Prior work on instruction prefetching has shown that instruction cache misses typically occur when execution transitions

[4]We note that SPEC FP workloads have an extremely small branch working set, with two benchmarks (519.lbm and 544.nab) having no L1-BTB misses at all.

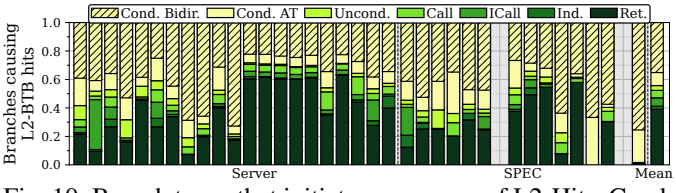


Fig. 10: Branch types that initiate a sequence of L2-Hits. Cond. Bidir.: conditional bidirectional, Cond. AT: conditional always taken branches

between code regions, such as when entering or returning from a function [27]. Given that the L1-BTB is an even more capacity-limited structure, likely to hold only the branches for a small active code region, such as a loop body or a function, we hypothesize that a similar phenomenon might apply to L1-BTB misses in that they are likely to occur in bursts corresponding to transitions from one code region to another.

To test this hypothesis, we analyze, for each fetch block that misses in the L1-BTB but hits in the L2-BTB, the preceding fetch block and classify whether it was an L1-BTB hit, an L2-BTB hit, an L2-BTB miss, or a non-existing entry – the latter referring to sequential code blocks without a taken branch that do not require a BTB entry (see Section VI for further details). We further distinguish whether the currently missing block lies on the predecessor's taken or not-taken (i.e., fall-through) path.

Fig. 9 confirms our hypothesis. The majority (76% on average) of L2-BTB hits are preceded by another L2-BTB hit, indicating that L1-BTB misses occur in clusters. The average length of L1-BTB *miss chains* that hit in the L2-BTB is 4.8 (not shown in the figure). These chains of L2-BTB hits tend to be immediately preceeded by an L1-BTB hit; fewer than 7% of L1-BTB misses occur after a block with no taken branch (which does not need a BTB entry) or an L2-BTB miss. Finally, we observe that the majority – 77%, on average – of predecessors of an L2-BTB hit lie on the taken path.

Take-away: L1-BTB misses tend to cluster into chains of back-to-back misses, nearly all of which hit in the L2-BTB. A typical miss chain is preceded by an L1-BTB hit.

C. What Triggers L1-BTB Miss Sequences?

Having established that L1-BTB misses occur in sequences, we next characterize the *triggers* that initiate them – these are the predecessor of the first L2-BTB hit in each sequence, corresponding to the L1-BTB hit category in Fig. 9. Based on our hypothesis that L1-BTB miss chains arise at code region transitions, we break down trigger branches by type: conditional Bidirectional (Cond. Bidir), conditional always-taken (Cond. AT)[5], unconditional, calls, indirect calls (ICalls), indirect, and returns.

Fig. 10 reveals that triggers are dominated by conditional branches and returns, accounting for 44% and 39% respectively, while direct and indirect calls together cause only 8.2% of triggers. The high dominance of returns may seem counterintuitive given calls are almost irrelevant – after all, each call has a corresponding return that transitions between code regions. The asymmetry stems from differing fan-out characteristics: a function is typically called from multiple call sites, increasing the likelihood that the function body is already resident in the L1-BTB by the time it is invoked. Returns, however, must transition back to these many distinct call sites, making it far less likely that the code region after the function call is tracked by the L1-BTB.

Among conditional branches, bidirectional branches are the dominant trigger of L1-BTB miss sequences, triggering 44% of all miss sequences. A more detailed analysis, omitted from the figure to avoid clutter, reveals that 2/3 of these bidirectional branches are attributable to loop exits , which further confirms that branches that transition execution from one region of code to another cause sequences of L1-BTB misses.

Take-away: Conditional bidirectional branches and returns trigger the majority of L1-BTB miss chains. The rest of the branches make up merely 17% of all triggers.

D. Summary of Insights

Our analysis reveals that the BTB must track a large active working set that eclipses the capacity of the L1-BTB by several orders of magnitude. Misses in the L1-BTB tend to occur in sequences, which we call "miss chains". Miss chains strongly correspond to transitions to new code regions such as when exiting a loop. The vast majority of all L1-BTB misses hit in the L2-BTB. The predecessor of an L1-BTB miss chain tends to be either an bidirectional branch (often corresponding to a loop exit) or a return.

IV. Toward a Practical L1-BTB Prefetcher

The huge disparity between the capacity of an L1-BTB that can be accessed in the single cycle and the size of the entire BTB working set suggests that prefetching (as opposed to, say, capacity management via a better replacement policy) is a promising approach. However, as we argue below, the large BTB working set presents a challenge for stateful prefetching approaches due to the large amount of metadata that such prefetchers would need to track. In response, we make a case for a metadata-free L1-BTB prefetching strategy and provide an intuition for how it might work. Section V then presents a detailed design that realizes the intuition with a low-complexity microarchitecture.

A. Strawman: a Stateful L1-BTB Prefetcher

Based on our characterization, which showed that misses occur in sequence (Section III-B) and that misses recur (Section III-A), a stateful prefetcher that learns miss sequences arises as an obvious solution. Potential candidates include Markov prefetchers as well as temporal streaming, which have been shown to be effective for data, instructions and TLB prefetching [25], [38], [56].

The drawback of these prefetchers, in the context of prefetching for the L1-BTB, is their high storage requirement, since they operate at the granularity of individual branches. In contrast, an instruction cache prefetcher tracks cache blocks

[5]Conditional branches that are never-taken are not tracked by the BTB

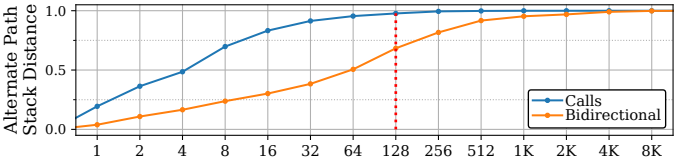


Fig. 11: Stack distance to alternate path instruction block.

(typically 64B), many of which are accessed sequentially, which means they can be naturally treated as spatial regions compactly encoded as bit-vectors. Similarly, an I-TLB prefetcher tracks entire memory pages (commonly 4KB or 2MB) with a single entry, naturally yielding a modest storage footprint. Moreover, whereas an instruction cache in today's high-end CPUs is typically 64KB or larger, thus justifying a storage-intensive prefetcher, the L1-BTB is necessarily much smaller given the need for single-cycle access latency.

For instance, tracking the misses of 10K branches using a Markov-based prefetcher, where missing BTB entries are associated with earlier trigger branches, would require tens of kilobytes of storage. For comparison, consider that a 128-entry BTB with 9 bytes per entry, would occupy slightly over 1 KB of storage, making it difficult to justify a stateful prefetcher with tens of KBs of storage. We refer the reader to Section VII-D for an in-depth comparison against such a stateful design.

B. Metadata-Free L1-BTB Prefetcher: the Intuition

Despite its simplicity, the analysis above suggests that a stateful prefetcher for an L1-BTB would need a significant amount of storage for high miss coverage. That makes it unattractive relative to the small size of the L1-BTB. Energy considerations for a stateful prefetcher would be yet another drawback of the approach.

The drawbacks of stateful prefetching motivate us to consider a metadata-free prefetcher. A naive approach might try to prefetch branches on the fall-through path; however, our characterization shows that the majority of L1-BTB misses are, in fact, on the taken path (refer to Fig. 9). Another naive method would be to exploit the insight that L2-BTB hits occur in chains and make a prediction for the next branch to be accessed upon an L2-BTB hit. Alas, that is precisely what the front-end's address-generation logic will do anyway, so the prefetch and demand path will coincide, rendering prefetching useless.

Our key insight, enabling an effective metadata-free design, relies on the fact that the vast majority of L1-BTB miss chains are triggered either by a bidirectional branch or by a return. We thus hypothesize that triggering prefetching on, what we refer to as, the *alternate path* of a branch instruction might be an effective way to anticipate upcoming L1-BTB misses. For a bidirectional conditional branch, the alternate path is the path contrary to the predicted one. For a call, it would be for the branches after the return.

To demonstrate a scenario where prefetching the alternate path might be helpful, consider the code example shown in Fig. 12. It describes a simple loop with an if-else statement.

```
LOOP_START:
        CMP     r1, r0
        BGE     LOOP_END        # Cond. Branch A (loop guard)
        CMP     r2, #0
        BEQ     ELSE_BRANCH     # Cond. Branch B (bidirectional)
IF_BRANCH:
        ...                     # Includes branches C, D, E
        B       LOOP_INCREMENT
ELSE_BRANCH:
        ...                     # Includes branches X, Y, Z
LOOP_INCREMENT:
        XOR     r2, r2, #1      # flip r2 every iteration
        ADD     r1, r1, #1
        B       LOOP_START
LOOP_END:
```
Fig. 12: Loop with an if-else statement.

Branch A serves as a loop guard and remains not-taken until the loop exits, while Branch B is a bidirectional conditional branch that switches direction each iteration. During the first iteration, Branch A is not-taken and Branch B is taken. The alternate path prefetcher will therefore prefetch the taken path of Branch A and the fall-through path of Branch B, yielding no immediate benefit in the first iteration. However, by the second iteration, when Branch B switches direction, the prefetcher will have brought in the entries on the fall-through path of Branch B (branches C, D, and E) covering potential BTB misses in the 'if' block. Additionally, once the loop exits, the prefetcher will have also covered the BTB entries along the loop-exit path (Branch A's taken path).

A critical question for the effectiveness of this approach is how soon the alternate path is executed after the branch or call is encountered, as a far distance risks the prefetched entries being evicted before they become useful. We answer this question by measuring the *BTB stack distance* between the BTB access to a bidirectional branch or call and the first BTB entry on the alternate path. For instance, for a loop guard branch, this would be the number of unique BTB entries touched within the loop body; for a call, it would be the number of unique BTB entries touched until (and including) the return.

Fig. 11 presents the stack distance separately for calls and all bidirectional branches. The results show that in 97% of cases, the return of a function call is within a stack distance of 128, while for 68% of bidirectional branches, fewer than 128 BTB entries are accessed before the first alternate path instruction executes.

Insight: in the large majority of cases, prefetching the alternate path can cover upcoming L1-BTB accesses even for a small L1-BTB capacity.

With this intuition in place, we next describe a simple microarchitectural realization of alternate path prefetching.

V. BTB-Ferret

A. High Level Overview

BTB-Ferret is a metadata-free L1-BTB prefetcher targeting alternate path BTB accesses of bidirectional conditional branches and calls. A BTB hit can trigger a chain of up to N BTB prefetches along the alternate path of the triggering branch: the opposite of conditional brnach predictor's (CBP)

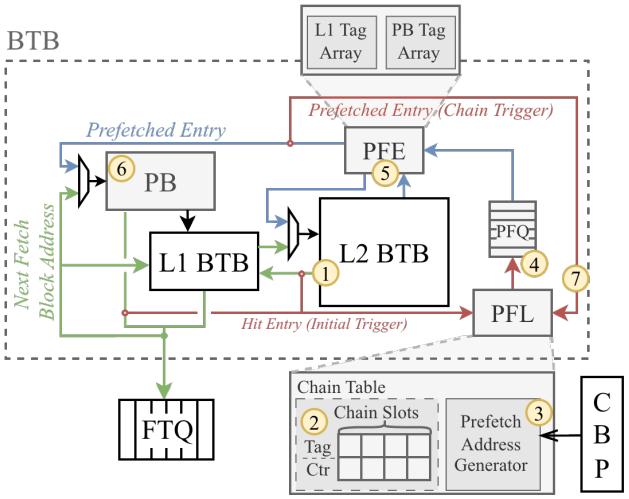


Fig. 13: BTB-Ferret design overview. New structures in grey.

prediction for bidirectional conditional branches, or the fall-through address of a call. This serves two purposes: to cover future executions of bidirectional conditional branches when they switch direction, and to prefetch the execution path when it inevitably returns upon function completion.

BTB-Ferret augments a two-level block-based BTB hierarchy with the components shown in grey in Fig. 13. The **Prefetch Logic** (PFL) is responsible for identifying branches that can trigger a prefetch chain, for generating the associated prefetch addresses, and for managing in-flight prefetch chains. The **Prefetch Queue** (PFQ) buffers prefetch addresses generated by the PFL until they are consumed by the Prefetch Engine. The **Prefetch Engine** (PFE) is responsible for prefetching BTB entries from the L2 to the L1-BTB while ensuring that prefetches for branches already present in L1-BTB or the Prefetch Buffer are dropped. To avoid L1-BTB pollution, the **Prefetch Buffer** (PB) stores prefetched entries until they are installed into the L1-BTB upon a demand access.

We continue this section with an operational example of a branch triggering a prefetch chain, and later explain the mechanisms of individual components in detail.

B. BTB-Ferret Operation

To illustrate the operation of BTB-Ferret, let A be the address of a fetch block containing a bidirectional conditional branch Br_A, predicted taken by the CBP. (1) Upon a demand access hit in the L2-BTB, A's BTB entry is passed into the Prefetch Logic (PFL). The PFL first checks the branch type and the *Alt* bit. The *Alt bit* is a BTB entry bit indicating whether the branch has previously changed direction. It is widely implemented in modern processors to enable the TAGE predictor to be queried only for branches that switch direction, thereby reducing power consumption [2]–[4]. Since Br_A is a bidirectional conditional branch, it is eligible to trigger a prefetch. (2) The PFL allocates a chain slot in the Chain Table with a unique *tag* and a *counter* set to N. (3) The PFL then uses the CBP prediction to generate the first prefetch address. As in this instance, the CBP predicts Br_A to be taken, the Prefetch Address Generator (4) enqueues the fall-through address (i.e., the opposite of the predicted path) with its corresponding *chain tag* into the Prefetch Queue (PFQ). Let B denote the fall-through address and Br_B the branch it contains.

Branch Type	Initial Trigger	Chain Trigger
Bidirectional Cond.	Opposite of CBP pred. [6]	Both Taken and Fall-Through Path
Always-Taken Cond.	None	Taken Path
Call (Direct/Indirect)	Fall-Through	BTB Target
Uncond. (Direct/Indirect)	None	BTB Target
Return	None	None

TABLE I: Prefetch Addresses for Branch and Trigger

Once address B reaches the front of the PFQ, (5) it is consumed by the Prefetch Engine (PFE). Before issuing the prefetch, the PFE first checks whether entry B is already present in the L1-BTB or the Prefetch Buffer (PB). If B is present, the prefetch is cancelled. Otherwise, the PFE reserves an entry in PB for B. Once prefetched, (6) entry B remains in PB until it is either evicted or a demand access fetches block B, in which case entry B is promoted to the L1-BTB.

Additionally, upon installation of entry B in the PB, (7) it is further fed back into the PFL to continue the prefetch chain. The corresponding *counter* in the Chain Table is decremented, and the Prefetch Address Generator generates a new prefetch address C, which is enqueued into the PFQ and subsequently prefetched (Section V-C explains in detail how chain prefetch addresses are generated). BTB entry C can similarly trigger further prefetches following the chain until the chain counter reaches 0.

C. Prefetching the Alternate Path

When a BTB entry is fed into the PFL, the Prefetch Address Generator attempts to generate a prefetch address. The prefetch address depends on the branch type and on whether the triggering branch initiates a new chain (*Initial Trigger*) or continues an existing one (*Chain Trigger*). A new chain can be initiated upon a hit in either the L2-BTB or the PB. The latter case ensures that a block prefetched into the PB (and that would otherwise constitute an L2-BTB miss) can itself serve as the starting point for a new prefetch chain.

Table I outlines the prefetch addresses generated for each branch type and trigger type. Upon an Initial Trigger, only the opposite of the predicted path is prefetched. As the chain continues, the Chain Trigger prefetches both the taken and fall-through paths for encountered bidirectional branches. The rationale for prefetching along both paths for branches encountered in the prefetch chain is to improve coverage.

D. Issuing Prefetches

The generated prefetch addresses are enqueued into the PFQ and consumed one by one by the Prefetch Engine (PFE). The PFE issues a prefetch only if the corresponding entry is not

[6]TAGE prediction is only available for prefetches triggered at L2 Hit (as TAGE and L2-BTB are accessed in parallel and have the same latency). For chains triggered on PB hit, we use the Bimodal prediction.

already present in the L1-BTB or the PB. Two implementation options exist for this filtering step. The first is to dual-port the L1-BTB and PB, allowing them to be directly consulted before issuing a prefetch. However, dual-porting these structures incurs additional area and energy overhead.

An alternative approach is to introduce compressed tag array copies of the L1-BTB and PB, with the same sizes, set associativities, and replacement policies. But instead of the full BTB target information, these tag arrays store only the compressed tags of the BTB entries. The level of tag compression determines the false hit and miss rates. The tag arrays are updated synchronously with each L1-BTB and PB read and write to keep them up-to-date about the main structures' contents.

Additionally, the PFE avoids issuing duplicate prefetches for entries that already have an in-flight prefetch request. This is achieved by reserving an entry in the PB prior to issuing the prefetch request to the L2-BTB. A further benefit of early reservation is that BTB-Ferret can reduce latency for late prefetches, which, although delayed, are still served faster than an on-demand L2-BTB access.

E. Chain Management

An *Initial Trigger* branch can initiate a chain of up to N prefetches. To curb Prefetch-Buffer (PB) thrashing, BTB-Ferret permits at most M chains to be alive at any moment. Their state lives in the *Chain Table*, whose M entries each hold a *chain tag* and a *chain counter*.

Slot allocation: Each time a new *Initial Trigger* arrives at the Prefetch Logic (PFL), it is assigned a Chain Table slot in a round-robin fashion, potentially evicting a currently active chain. The *chain counter* is initialized to N upon chain creation and decremented each time a Chain Trigger corresponding to that chain arrives at the PFL.

Tag and index generation : Chain tags are assigned in increasing order according to newTag $=$ (prevTag $+ 1$) $\bmod (M \times 2)$, which makes the Chain table indexing straightforward (index $= \lfloor \text{tag}/2 \rfloor$) and minimizes tag storage overhead while making tag aliasing rare.

Tag Propagation: When the generated prefetch address is enqueued into the PFQ, it is tagged with its *chain tag*. PFE propagates the tag along with the prefetch request so that upon arrival of the corresponding Chain Trigger in PFL, the correct Chain Table slot can be identified and updated.

Chain extension: Any *Initial* or *Chain Trigger* can advance prefetching along the chain further, provided that the following conditions are met: (1) chain counter is greater than 0, (2) the corresponding entry is present in the L2-BTB but not in the L1-BTB, and (3) the branch is not a return.

VI. Methodology

Workloads We run a wide range of typical server and SPEC workloads from six different benchmark suites, summarized in Table II. These comprise 16 Java benchmark workloads [9], [14], [54], two web services (NodeApp [17] and PHPWiki [21]). Additionally, we included 3 workloads from Google's Fleetbench benchmarks suite [20], a set of relevant kernels that contribute a significant portion of execution cycles in Googles data centers [1]. Finally, we also include 16 SPEC workloads to capture a range of more traditional CPU workloads. Since many applications are JIT-compiled, we ran each workload for several minutes using gem5's KVM core to ensure we evaluated optimized, steady-state code. Once the application enters a stable state, we take a checkpoint and perform a SimPoint analysis [53] for the stable region. Based on the analysis, we use the most representative SimPoints from 200M instructions (100M warmup) for each workload in our evaluations. The workloads and simulation infrastructure is available at https://github.com/yongjiehuang/BTB-Ferret.

Application	**Description**
NodeApp	NodeJS webserver
PHPWiki	PHP wiki web server
Proto, TCMalloc, STL	Google Fleetbench [1]
H2, H2O, Luindex, Spring, Tomcat, Lusearch	Java DaCapo suite [9]
Voter, Twitter, TATP, Epinions, YCSB, Seat, SiBench, Noop, SmallBank	Java BenchBase suite [14]
Finagle-HTTP	Java Renaissance suite [54]
500.perlbench, 502.gcc, 505.mcf, 520.omnetpp, 523.xalancbmk, 525.x264, 531.deepsjeng, 541.leela, 557.xz	SPECINT [13]
508.namd, 510.parest, 511.povray, 519.lbm, 526.blender, 538.imagick, 544.nab	SPECFP [13]

TABLE II: Workloads used to evaluate BTB-Ferret

Core	3GHz, 8-way OoO, 576 ROB, 190/120 LQ/SQ
Branch Predictor	Base: 8K entry Bimodal, 1-cycle Overriding: 64KiB TAGE-SC-L [51], 3-cycles Indirect: 64KiB ITTAGE [49], 3 cycles
BTB	L1-BTB: 128 entry, 8-way, 16-tags, 1-cycle, L2-BTB: 16K entry, 8-way, 16-tags, 3-cycles, mostly exclusive
Caches	L1-I: 64KiB, 16-way, 1 cycle[7], 10 MSHRs L1-D: 48KiB, 12-way, 4 cycle, 16 MSHRs L2: 3MiB, 12-way, 17 cycle, 32 MSHRs LLC: 8MiB, 16-way, 51 cycle, 64 MSHRs
Prefetchers	Instructions: FDIP + NL, Data: BOP [37], L2: Next-line
Memory	DDR4 2400MHz, 12.5 ns RCD/RP/CAS

TABLE III: Parameters of the simulated processor.

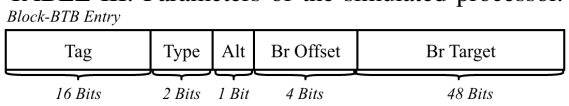


Fig. 14: Block BTB Entry Structure.

Simulator Infrastructure: To evaluate BTB-Ferret we use gem5 [8], [18], [36] v25.0.0.0, allowing detailed cycle-accurate, full-system simulations. We integrated patches for the decoupled front-end [42], [47], fixes for the speculative history update of TAGE-SC-L in gem5 [43], and the IT-TAGE [44], [49]. We configure gem5 after an Intel Granite Rapid [32], [34] with the parameters detailed in Table III, reflecting a high-performance industry baseline.

To drive the decoupled front-end, we model a block-based BTB indexed by a block's start address rather than the branch PC [40], as required by FDIP to identify the next branch

[7]Since gem5 does not support a micro-op cache, the L1-I cache is configured with the micro-op cache latency, instead of the L1-I cache latency as described in [22].

without searching every valid instruction address [40], [41]. A block is created on first execution and spans a sequential code region up to the next always-taken or bidirectional branch, which determines the block's end. Never-taken branches within a block do not consume BTB storage, as they are invisible to the BPU and treated as regular instructions. Blocks are capped at 64B; if no taken branch exists within a 64B region, the fetch PC advances by the fixed 64B offset, and subsequent block start addresses are aligned accordingly. This design reflects commercial BTB implementations [3]. The entry organization of our BTB is visualized in Fig. 14.

We extend gem5 with a two-level branch predictor hierarchy in which TAGE-SC-L overrides its bimodal base component after three cycles, and a two-level BTB hierarchy consisting of a 128-entry L1 backed by a 16K-entry L2 acting as a victim cache. The processor interacts exclusively with the L1-BTB. Only on an L1-BTB miss, the L2-BTB is accessed and, upon a hit, the entry is copied back into the L1-BTB. A new entry is created by displacing a victim entry from the L1 entry into the L2-BTB using LRU replacement before allocating the new branch itself in the L1-BTB.

A key modelling objective is correctly capturing override behaviour. Since gem5 resolves conditional predictions and BTB lookups instantaneously, it is known at prediction time whether TAGE or the L2-BTB will cause an override. A straightforward approach would be to stall the generation of new fetch targets for the corresponding number of cycles. While this correctly models front-end bubbles, it fails to capture false-path prefetching: during an override, the CPU continues issuing fetch targets along the predicted path, driving speculative instruction prefetches into the FTQ. We find that these "override" prefetches significantly affect FDIP's performance, and omitting them distorts the results. We therefore model overrides by continuing to generate fetch targets using the L1-BTB and bimodal predictions to faithfully reproduce false-path prefetching behaviour during override.

Simulated BTB designs:

***Baseline*:** The baseline in this work is a two-level block-based BTB hierarchy with a 128-entry, 1-cycle L1-BTB and a 16 K-entry, 3-cycle L2-BTB that is accessed serially after an L1-BTB miss, yielding a 4-cycle total latency for an L2-BTB hit.

Markov: We compare BTB-Ferret against a Markov prefetcher as a strong stateful baseline. Upon an L2-BTB or PB hit, the hitting PC indexes the Markov table whose size we vary from 512 to 4K entries, each holding 1-4 previously recorded successor addresses, all of which are enqueued into the PFQ for prefetching. We assume each entry consists of a 16-bit tag and N successor fields, each comprising a 64-bit prefetch PC and a 2-bit frequency counter, resulting in $16+N\times 66$ bits per entry. Thus, for the evaluated Markov configurations of 512-4K entries, the required table capacity is 5-41KiB, 9.3-74KiB, and 17.5-140KiB for 1, 2, and 4 successors, respectively. We further assume an ideal 0-cycle Markov table lookup latency.

The Markov table is populated by associating L2-BTB hits with an earlier L2-BTB miss. To cover the L2-BTB access latency, we associate not the immediate predecessor but the 4th preceding BTB entry, assuming one entry is processed per cycle. Entries in the Markov table are replaced using LRU, while successors are replaced by frequency: each time a successor is observed missing in the L2-BTB, its counter is incremented.

Akin to BTB-Ferret, Markov prefetches are triggered on L2-BTB and prefetch hits. However, in contrast to BTB-Ferret, which performs additional chain prefetches upon each prefetch fill in the PB, the Markov prefetcher records multiple successors per entry, allowing it to capture multiple branches of the chain and issue all prefetches back-to-back.

***BTB-Ferret*:** Our proposal uses a 32-entry prefetch buffer (PB) and an 8-entry prefetch queue (PFQ). A PB entry is equivalent to an L1-BTB entry, as shown in Fig. 14, with an additional valid bit to mark whether the prefetch has already arrived or is still in flight. The PFQ entry consists of a 48b prefetch address and the 5-bit chain ID. The total size of the PB and PFQ is 288 and 52 bytes, respectively.

BTB-Ferret tracks 8 chains in parallel with a maximum number of 14 prefetches per chain (chain counter = 4 bit). The two upper bits of the chain ID serve as a tag to mark each chain, resulting in a total capacity of 48 bits for the chain table.

We empirically found 12-bit tags for the secondary tag arrays to be a good trade-off between storage and false positive rate, yielding a 0.04% false positive rate with no noticeable impact on performance.

Thus, the secondary arrays for the L1-BTB and PB consume 240 bytes, bringing the total storage of BTB-Ferret to 586 bytes.

We assume one cycle to compute new prefetch addresses and filter entries already present in the L1-BTB or PB. Only one prefetch or demand access can be issued per cycle (demand accesses have precedence), meaning that in case BTB-Ferret wants to issue multiple prefetches, it must do so over multiple cycles.

VII. Evaluation

A. Performance

We first evaluate the performance improvements BTB-Ferret achieves over the 2-level Baseline across our workload suit, using an idealized single-cycle L2-BTB as an upper-bound reference. Results are presented in Fig. 15, normalized to a 2-level BTB baseline without prefetching into the L1-BTB.

BTB-Ferret consistently delivers a performance improvement with an average IPC uplift of 1.7%, and maximum of 9.7% on 531.deepsjeng. Relative to an idealized single-cycle BTB, BTB-Ferret captures 48.5% of the opportunity, demonstrating the effectiveness of its metadata-free organization. Performance gains track well with the opportunity across the entire workload suit. On SPEC, performance gains average 2.4%; on server workloads, 1.1%.

The performance gains can be explained by two main effects. First, by reducing L1-BTB misses, BTB-Ferret allows the BPU to maintain sufficient runahead distance, reducing

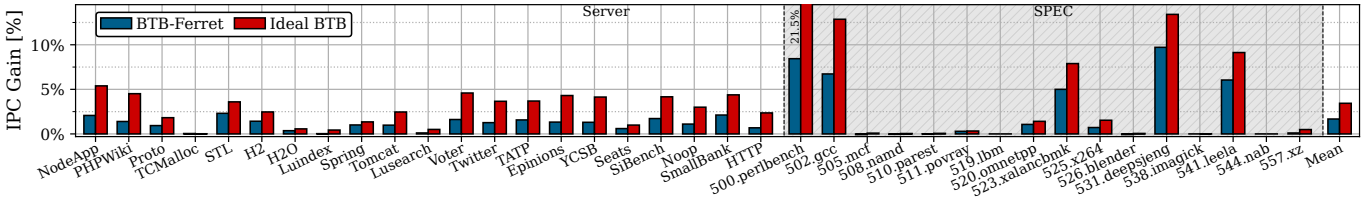


Fig. 15: Performance of BTB-Ferret compared to the Ideal 1-cycle L2-BTB

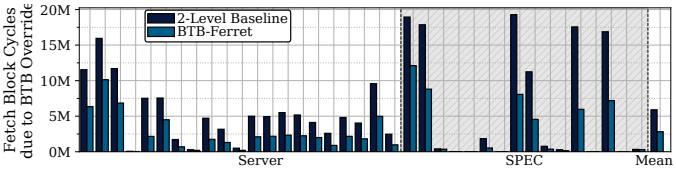


Fig. 16: Cycles fetch is blocked due to L1-BTB overrides

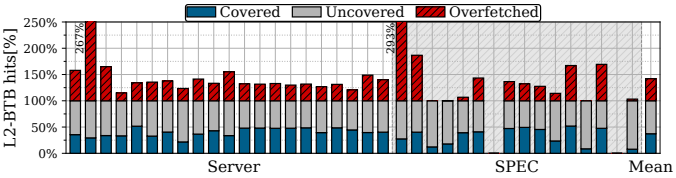


Fig. 17: Fraction of L2-BTB hits covered and incorrectly predicted by BTB-Ferret. 519.lbm and 544.nab have no L1-BTB misses and therefore don't trigger prefetches.

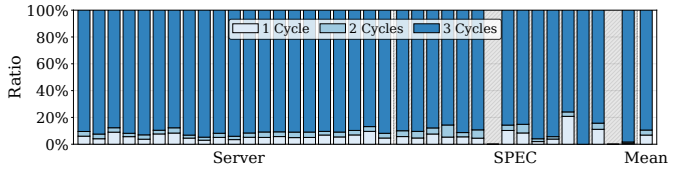


Fig. 18: Fraction of covered cycles by BTB-Ferret

the BPU latency exposed to the fetch unit. Fig. 16 plots cycles where the front-end is blocked due to a BTB override, showing that BTB-Ferret reduces these exposed cycles by 46%. Second, the larger runahead distance improves FDIP's prefetching timeliness. Indeed, we find BTB-Ferret reduces L1-I misses by 21% and I-cache stall cycles by 15% on average (not shown in the graph).

B. Coverage and Accuracy

Next, we study the effectiveness of BTB-Ferret in covering L1-BTB misses that hit in the L2-BTB. We categorize L2-BTB hits as (1) covered by BTB-Ferret (i.e., hit in the PB), (2) uncovered, or (3) overprefetched (i.e., prefetched into the PB but never used). Fig. 17 shows this breakdown relative to all L2-BTB hits[8].

BTB-Ferret covers an average of 37.3% of all L1-BTB misses hitting in the L2-BTB, closely mirroring the performance opportunity observed in Fig. 15. Coverage ranges from 8.0% for 557.xz to 51.9% for 531.deepsjeng. Since BTB-Ferret allocates a PB entry upon issuing a prefetch, a demand access may hit the PB before the entry has fully arrived, resulting in only partial latency coverage. Fig. 18 further breaks down covered accesses by the number of cycles covered, revealing that the vast majority of prefetches (75.9–100%, avg. 89.4%) cover the full 3-cycle L2-BTB access latency, with only 10.6% arriving partially late, confirming that BTB-Ferret's prefetches are timely.

In terms of overprefetch, Fig. 17 shows that BTB-Ferret increases L2-BTB accesses beyond L2-BTB hits by 42% on average. For most benchmarks, the overhead is moderate, but PhpWiki and 500.perlbench are outliers at 166.8% and 192.5% additional accesses respectively, skewing the average.

We performed a root-cause analysis to understand what causes useless prefetches and found that most (68%) are triggered by conditional branches, in particular branches with a strong bias toward one direction (37% of all useless prefetches). For these mostly-taken or mostly-not-taken branches, BTB-Ferret tends to prefetch the path that is often not useful, whereas the demand execution follows the typical (biased) path. While we show in our energy evaluation (Section VII-C) the energy implications of overfetch are tolerable, there is a natural opportunity for future work to selectively suppress prefetching for biased branches, potentially at the cost of additional monitoring state.

C. Energy Estimation

As the BTB steers the entire CPU pipeline, BTB-Ferret affects core energy in two opposing ways. On the one hand, useless prefetches and PB accesses increase the energy of the BTB complex itself. On the other hand, fewer L1-BTB misses reduce BTB overrides, leading to fewer execution cycles, fewer false-path instruction prefetches, and fewer FTQ squashes, all of which reduce overall core energy.

To understand these two effects, we measure read accesses to the L1- and L2-BTB. The L1-BTB is accessed every cycle that the BPU is not blocked (e.g., because of a full FTQ or back-end pressure), while the L2-BTB is accessed on every L1-BTB miss or BTB-Ferret prefetch. We break down accesses into normal accesses and *override accesses* – i.e., accesses generated in the cycles after the L1-BTB experiences a miss and until it is corrected by the L2. The override accesses are detrimental to both performance and energy-efficiency as they require accesses to the L2-BTB and trigger likely-useless instruction prefetches.

Fig. 19 shows the average L1-BTB and L2-BTB accesses for BTB-Ferret compared to the baseline. We find that BTB-Ferret reduces L1-BTB accesses by 9.5%, thanks to a 20.7%

[8]Note that the total number of L2-BTB accesses is higher as not all accesses result in an L2-BTB hit (i.e. L2-BTB misses and instruction blocks without a BTB entry)

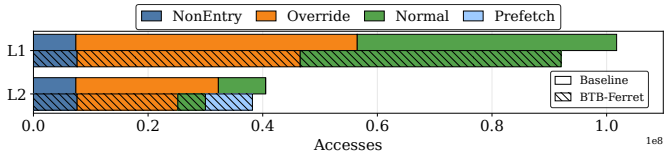


Fig. 19: Break down of L1-BTB and L2-BTB accesses

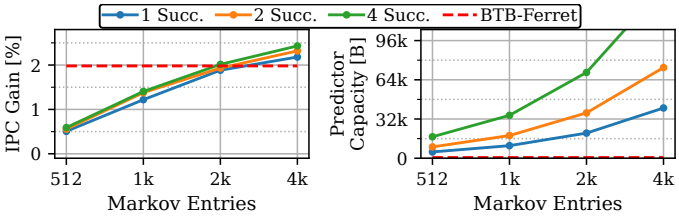


Fig. 20: BTB-Ferret compared to a Markov prefetcher

decrease in override accesses. The reduction in overrides is even more relevant for L2-BTB accesses. As the breakdown reveals, 61.3% of all L2-BTB accesses are initiated during an override. This makes intuitive sense: once the BPU strays off path into a region not cached in the L1-BTB, subsequent accesses will likely also miss until the L2-BTB override sets it back on the correct path. By increasing L1-BTB hits, BTB-Ferret reduces override-driven L2-BTB accesses by 29.3%. Interestingly, this reduction outweighs the 42% increase in accesses due to overprefetching (relative to normal accesses), resulting in a net reduction in L2-BTB accesses of 5.7%.

Using the access statistics from Fig. 19 and CACTI 7.0 [7], we compute the BTB complex's access energy. Despite fewer overall BTB accesses, BTB-Ferret increases the access energy of the BTB complex by 18%, driven by the additional accesses to the PB and the duplicate tag arrays. Note that this access energy estimate does not capture the energy savings from fewer overrides due to a reduction in (1) wrong-path instruction prefetches and (2) FTQ enqueues and dequeues on the overriden path.

D. BTB-Ferret Against a Stateful Prefetcher

A key feature of BTB-Ferret is its stateless design, well-suited to the constrained environment of the L1-BTB. To contextualize its efficiency, we compare BTB-Ferret against a stateful Markov-based prefetcher [23], [39], [56]. For the Markov prefetcher, we ignore all storage, latency, and area constraints, thus providing a strong comparison point against metadata-free BTB-Ferret. We refer to Section VI for the Markov prefetcher's implementation details.

The two key design parameters governing Markov prefetcher coverage and hardware overhead are the number of trigger table entries and the number of successor prefetches per trigger. To characterize the performance and storage trade-off, we sweep over 512 to 8K table entries and 1, 2, and 4 successors[9]. Fig. 20 presents the resulting performance improvements and storage requirements.

As expected, more table entries allow the Markov prefetcher to track more trigger PCs, yielding higher performance. With

[9]Using 8 and 16 successors didn't yield further performance gains.

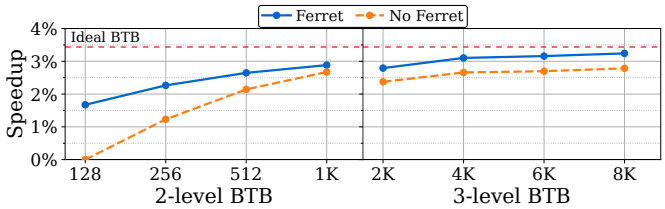


Fig. 21: Sensitivity of BTB-Ferret to L1-BTB size (left) and three-level BTB hierarchies (right).

4 successors and 512 entries, Markov achieves 0.5% performance improvement, reaching 2.5% with 4K entries. To match BTB-Ferret's performance, a Markov configuration of at least 2K entries with 2 successors is required, incurring a storage cost of 29KB. BTB-Ferret achieves the same performance with only 587B – less than 2% of the storage overhead of the equivalent Markov configuration – demonstrating that its stateless design delivers competitive performance at a fraction of the cost.

E. BTB-Ferret in Different BTB Organizations

So far, we have only considered a two-level BTB configuration with a 128-entry L1-BTB and 16K-entry L2-BTB. Here, we consider the broader design space of (1) larger L1-BTB capacities, and (2) three-level BTB hierarchies, seeking to establish whether BTB-Ferret continues to be beneficial in these settings.

L1-BTB size. In the left part of Fig. 21, we evaluate performance given L1-BTB capacities ranging from 128 to 1K entries with and without BTB-Ferret. The figure, showing average speedup normalized to the 128-entry baseline, reveals that a larger L1-BTB reduces the opportunity for prefetching, narrowing the performance gap with ideal to 0.8% at 1K-entries. However, even with larger L1-BTB, BTB-Ferret continues to deliver performance gains of 1%, 0.5%, and 0.2% at L1-BTB capacities of 256, 512, and 1K-entries, respectively. Importantly, the sweep reveals that BTB-Ferret delivers the same or greater performance gains compared to doubling the L1-BTB capacity, at almost zero area cost. For instance, a 256-entry L1-BTB with BTB-Ferret improves performance by 2.3%, compared to 2.1% achieved by a 512-entry L1-BTB without BTB-Ferret.

Three-level BTBs. In the right part of Fig. 21, we evaluate BTB-Ferret in a three-level hierarchy with an additional mid-level BTB. We sweep the size of this mid-level BTB from 2K to 8K entries and show average speedups over the two-level baseline.

The L1-BTB and L3-BTB configurations exactly match the two-level baseline, with 128 and 16K entries and access latencies of 1 and 3 cycles, respectively. For the mid-level BTB, we model a 2-cycle lookup latency and a parallel access with the L1-BTB, resulting in 1 cycle of exposed latency on an L1-BTB miss and mid-level BTB hit. Note that this setup aligns with Intel's Lion Cove BTB organization [32].

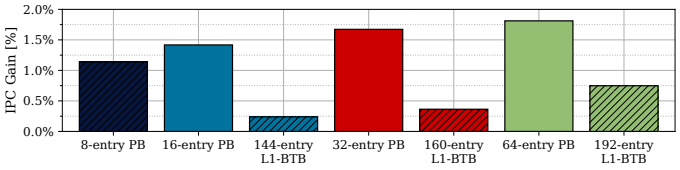


Fig. 22: Sensitivity to the PB size with a 128-entry L1-BTB. Comparison points are iso-capacity L1-BTB designs.

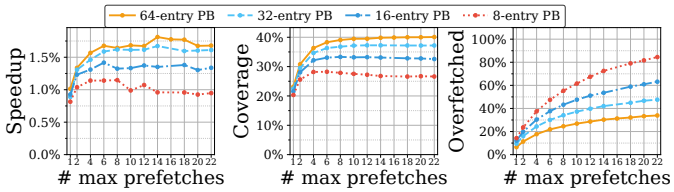


Fig. 23: Sensitivity to the maximum prefetches per chain.

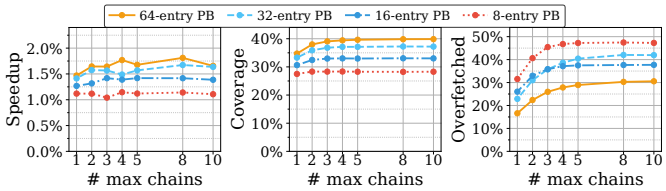


Fig. 24: Sensitivity to the maximum in-flight chains

The results show that while a third level partially closes the gap to the ideal BTB, BTB-Ferret still manages to improve performance. For instance, a 6K-entry mid-level BTB with BTB-Ferret delivers 3.2% performance over the two-level baseline, compared to 2.7% without BTB-Ferret. Furthermore, similar to the L1-BTB sensitivity above, BTB-Ferret enables smaller BTB capacities without losing performance. For instance, combining BTB-Ferret with a 2K-entry mid-level BTB achieves the same effect as an 8K mid-level, at a quarter of the latter's area cost.

F. Prefetch Buffer Sensitivity

We study the sensitivity of BTB-Ferret to the prefetch buffer (PB) size. We evaluate 8, 16, and 32-entry PB configurations paired with a 128-entry L1-BTB, and compare against an equivalently sized L1-BTB to provide a fair baseline. We model the 144-entry (128+16) and 160-entry (128+32) L1-BTB by increasing associativity from 8 to 9 and 10 respectively; no log2-compliant configuration exists for 136 entries, so this data point is omitted from the figure.

Fig. 22 presents average performance improvements over the 2-level baseline. Reducing the PB from 32 to 16 and 8 entries reduces BTB-Ferret performance improvement from 1.7% to 1.4% and 1.1%, respectively, showing that even a small PB captures a significant fraction of the opportunity. Crucially, BTB-Ferret consistently outperforms an equivalently sized L1-BTB: while a 160-entry L1-BTB improves performance by only 0.4%, a 32-entry PB delivers a 4.3x higher improvement of 1.7%, confirming that prefetching is far more effective than simply enlarging the L1-BTB.

G. Chain Sensitivity

Finally, we evaluate the sensitivity to the Chain Table configuration. Specifically, we study how the number of prefetches per chain (prefetch depth) and the number of parallel chains (inflight chains) affect the performance of BTB-Ferret.

Maximum prefetches per chain. For this study, we fix the number of inflight chains to 8, which we found high enough to almost never be exceeded ($< \%1$ of the execution time).

Fig. 23 illustrates the sensitivity of BTB-Ferret to the maximum prefetches per chain on four different sizes of PB. Increasing the prefetch depth initially improves coverage by allowing BTB-Ferret to look further ahead, which translates directly into higher performance. With 32- and 64-entry PBs, speedup peaks at 14 prefetches per chain, whereas the 8- and 16-entry PBs achieve their best performance at 6. However, a longer chain also increases overprefetches, which diminishes coverage benefits and negatively affects performance by kicking out prefetched entries before they are used (timeliness issue). This trade-off is most pronounced for the 8-entry PB: beyond 6 prefetches per chain, coverage begins to decrease while overprefetching continues to rise. Larger PBs can retain a prefetched entry longer, thereby being more tolerant to deeper prefetch chains. We also found that the max number of prefetches is often not reached because a chain terminates early when a prefetch in chain is canceled, e.g., it is already located in the L1-BTB/PB.

Maximum in-flight chains. To evaluate the sensitivity on the parallel chains we fix the max prefetches per chain to the best value found in Fig. 23 (14 for 64- and 32-entry PBs, and 6 for 8- and 16-entry PBs.) Fig. 24 presents the effect of BTB-Ferret with different sizes of the Chain table. It reveals that BTB-Ferret is relatively insensitive to the number of inflight chains. Going from 1 to 4 parallel chains improves coverage by only 1-5% across the evaluated PB sizes. This insensitivity stems from two key factors: (1) Early termination of a chain shortens the duration of a table slot occupied by an inflight chain; (2) The low temporal density of initial triggers makes BTB-Ferret rarely trigger multiple chains within a short execution window.

VIII. Related Work

Our work targets prefetching in hierarchical BTBs similar to the ones that exist in contemporary high-end CPUs [12], [32], [33], featuring a small single-cycle L1-BTB backed by large but slow additional BTB level(s). An important goal of such hierarchical BTBs is maintaining high BTB throughput of at least one target per cycle. To the best of our knowledge, existing research has not studied such BTB organizations.

Conceptually, the closest work to ours describes bulk prefetching of BTB entries in a 2-level BTB hierarchy of IBM's Z-series EC12 processor [10]. There are large differences between that work and ours due to extremely different BTB organizations and throughput requirements. The L1-BTB in the IBM processor tracked 4K branches using a spatial organization indexed by the program counter (PC); finding the next branch on the fall-through path required searching the

BTB over a spatial region by incrementing the PC by the size of the instruction word. Thus, the BTB had low throughput (1 branch per multiple cycles) and could not distinguish the case of a PC not being a branch from a true BTB miss. The BTB prefetcher was thus optimized for such serial search in the L1-BTB by prefetching a large number of branches that lie in a contiguous region of code from the L2-BTB into a massive prefetch buffer capable of holding over 700 branches. In contrast, our work targets a high BTB throughput scenario where the L1-BTB produces a target every cycle, which mandates a small yet fast L1 structure and precludes bulk preload of many entries at once. Our BTB hierarchy is modeled after today's processors, which do not organize the BTB into spatial regions thus precluding spatial preloading used by EC12.

Phantom BTB [11] focuses on increasing the BTB capacity by introducing a high-capacity BTB level and virtualizing it into the LLC. To hide the access latency to the virtualized BTB, PhantomBTB introduces a temporal stream-based prefetcher. The prefetcher requires a considerable amount of metadata (256 KB of metadata for a virtualized BTB capacity of 210 KB). In contrast, BTB-Ferret targets prefetching into the small L1-BTB with no additional metadata beyond what already exists in today's BTBs.

Another class of BTB prefetchers, including Confluence [26], Shotgun [27], and Boomerang [28], prefetch both instructions and BTB metadata simultaneously. Similar to Phantom BTB, these works do not focus on the L1-BTB hit rate but rather on prefetching into a multi-Kentry single-level BTB.

AVM-BTB [35] is a rare work that does model a hierarchical BTB. However, its focus is on BTB capacity expansion by dynamically turning the μop cache and part of the L1-I cache into an additional BTB structure when the L2-BTB is under pressure. AVM-BTB does not improve the L1-BTB hit rate.

A recent work studies the organization of BTB entries [40]. While the work models a BTB hierarchy, its contribution lies in the exploration of advantages and drawbacks of various BTB entry types. It neither implements a BTB prefetcher nor seeks to improve the L1-BTB hit rate.

IX. Conclusion

Our work shines a light on the importance of L1 structures in hierarchical BPUs, common in today's high-performance CPUs. Our characterization shows that massive branch working sets overwhelm the L1 structures. The L1-BTB is a particular pain point due to its importance for instruction prefetching via the FTQ. We identify L1-BTB misses as symptoms of code-region transitions and propose BTB-Ferret, a metadata-free prefetch scheme, that anticipates such transitions and prefetches upcoming BTB entries into a compact prefetch buffer. A key finding of our work is that high-coverage L1-BTB prefetching is possible without storage-hungry machinery. More broadly, this work opens the door to future studies into hierarchical BPUs, and our open-source gem5 modifications facilitate it.

Acknowledgment

The authors thank the anonymous reviewers, the members of the Systems Research Group at the Technical University of Munich, and the EASE Lab team at the University of Edinburgh for their valuable feedback on this work. We also thank the Systems Research Group for supporting Yongjie Huang's remote internship at the Technical University of Munich and for providing the computing infrastructure used to conduct the experiments for this work. This research was generously supported by the University of Edinburgh and by EASE Lab's industry partners and sponsors, including ARM, Huawei, and Intel.

Appendix

A. Abstract

This artifact presents the code and methodology for BTB-Ferret, available at https://github.com/yongjiehuang/BTB-Ferret/, containing a gem5 implementation of BTB-Ferret.

The artifact is intended to support future research by documenting how to build and evaluate BTB-Ferret with gem5. While it illustrates the full workflow for the simulation—building and running BTB-Ferret model, and analyzing results—it does not aim to reproduce every result presented in the paper.

B. Artifact check-list (meta-information)

- **Compilation:** C++ compiler with C++20 standard.
- **Model:** BTB-Ferret models for gem5.
- **Metrics:** Normalized IPC and prefetch coverage
- **Experiments:** Use the provided script to evaluate IPC improvement over the baseline BTB hierarchy without BTB-Ferret and the reduction of L1-BTB misses.
- **How much disk space required (approximately)?:** $\sim$20GiB for building gem5 and run the experiments + $\sim$40GiB for full-system setup with Simpoint checkpoints.
- **How much time is needed to prepare workflow (approximately)?:** $\sim$20 min. Mostly to build gem5.
- **How much time is needed to complete experiments (approximately)?:** Each configuration alone takes between 17 and 250 minutes to simulate, depending on the the benchmark, and the used machine. Running all 3 configurations each with 37 apps on a 64-core machine should take about 5 hours. Building gem5 takes another $\sim$20 min.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.21478670

C. Description

1) How to access: The source code is publicly available on GitHub (https://github.com/yongjiehuang/BTB-Ferret/) or Zenodo (https://doi.org/10.5281/zenodo.21478670).

2) Hardware Dependencies: The BTB-Ferret framework can be used on any system with a general-purpose CPU and at least 60 GiB free disk space to store the apps checkpoints and set up gem5 full-system simulation.

3) Software Dependencies: The framework is tested on Ubuntu 22.04 and 24.04 with GCC v11.4.0 and Clang v11.0 and v14.0. For plotting the graphs with Python 3.10, the matplotlib along with pandas and numpy library are used.

Running gem5 experiments requires all dependencies to build gem5. Refer to the gem5 documentation for details [19]

D. Installation

This section explains how to obtain the source code and build the gem5 simulator. Because setting up full–system simulations with complex server workloads is beyond the scope of this artifact—and is already documented extensively in prior work [19], [29], [46]—we do not provide the full system setup used in our evaluation.

We assume Nix is available, otherwise, refer to the gem5 documentation [19] to set up all dependencies required to build gem5 on your system. To prepare the gem5 simulation and build the sources, run the following steps.

```
# Clone the BTB-Ferret repository
git clone git@github.com:yongjiehuang/BTB-Ferret.git
cd BTB-Ferret
# Enter the Nix shell
nix-shell default.nix
# Build gem5
cd gem5
scons build/ARM/gem5.opt -j $(nproc)
cd ..
```

E. Simulation

To reproduce the IPC improvement gained by BTB-Ferret, similar to Fig. 15 as well as the Coverage of BTB-Ferret illustrated by Fig. 17, use the following commands to launch the simulations for the 2-level baseline BTB hierarchy, BTB-Ferret and the Ideal BTB hierarchy, respectively. Note that each line invokes 37 gem5 processes in parallel, as we evaluate BTB-Ferret on 37 benchmarks in this work.

```
./simpoints/all_4_run_single_simpoints.sh --baseline
    -2level
./simpoints/all_4_run_single_simpoints.sh --BTB-
    Ferret
./simpoints/all_4_run_single_simpoints.sh --Ideal-
    BTB
```

F. Evaluation and Expected Results

The `./results` folder should contain 3 experiments directories, corresponding to 3 configurations simulated, each with five files per benchmark. Use the plotting script to parse the files and evaluate the results.

```
python eval_l1main.py
```

This should produce two PDFs plotting performance of BTB-Ferret and the prefetch coverage, should be similar to Fig. 15 and Fig. 17.

G. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

References

[1] A. Abel, Y. Li, R. O'Grady, C. Kennelly, and D. Gove, "A profiling-based benchmark suite for warehouse-scale computers," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2024, pp. 325–327.

[2] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito, "The ibm z15 high frequency mainframe branch predictor industrial product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2020, pp. 27–39.

[3] I. Advanced Micro Devices, "Software optimization guide for the amd zen4 microarchitecture," Advanced Micro Devices, Inc., Cambridge, MA, USA, Tech. Rep., 2023.

[4] I. Advanced Micro Devices, "Software optimization guide for the amd zen5 microarchitecture," Advanced Micro Devices, Inc., Cambridge, MA, USA, Tech. Rep., 2024.

[5] T. Asheim, B. Grot, and R. Kumar, "A storage-effective btb organization for servers," *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1153–1167, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:255570210

[6] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers." ACM, 2019, pp. 462–473.

[7] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, pp. 14:1–14:25, 2017.

[8] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. S. B. Altaf, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[10] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. R. Prasky, and A. Saporito, "Two level bulk preload branch prediction," *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 71–82, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:17586579

[11] I. Burcea and A. Moshovos, "Phantom-btb: a virtualized branch target buffer design," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, M. L. Soffa and M. J. Irwin, Eds. ACM, 2009, pp. 313–324. [Online]. Available: https://doi.org/10.1145/1508244.1508281

[12] B. Cohen and M. Subramon. (2024) Next generation "zen 5" core. [Online]. Available: https://hc2024.hotchips.org/assets/program/conference/day2/24_HC2024.AMD.Cohen.Subramony.final.pd

[13] S. P. E. Corporation. (2017) Spec cpu 2017 benchmark. [Online]. Available: https://www.spec.org/cpu2017/

[14] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *PVLDB*, vol. 7, no. 4, pp. 277–288, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[15] DynamoRIO. (2024) Google workload traces. [Online]. Available: https://dynamorio.org/google_workload_traces.html

[16] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 152–162, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:9228401

[17] O. Foundation. (2024) Introduction to node.js. [Online]. Available: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

[18] gem5 developers. (2022) gem5. [Online]. Available: https://github.com/gem5/gem5/releases/tag/v22.0.0.1

[19] gem5 developers. (2025) gem5 documentation. [Online]. Available: https://www.gem5.org/documentation/

[20] Google. (2026) Fleetbench. [Online]. Available: https://github.com/google/fleetbench

[21] T. P. Group. (2024) Fastcgi process manager (fpm). [Online]. Available: https://www.php.net/manual/en/install.fpm.php

[22] U. D. C. A. R. Group. (2020) gem5 skylake config. [Online]. Available: https://github.com/darchr/gem5-skylake-config/blob/master/configuration-details.md

[23] D. Joseph and D. Grunwald, "Prefetching using markov predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 252–263. [Online]. Available: https://doi.org/10.1145/264107.264207

[24] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer." ACM, 2015, pp. 158–169.

[25] C. Kaynak, B. Grot, and B. Falsafi, "Shift: Shared history instruction fetch for lean-core server processors," *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 272–283, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:8077124

[26] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: unified instruction supply for scale-out servers," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, M. Prvulovic, Ed. ACM, 2015, pp. 166–177. [Online]. Available: https://doi.org/10.1145/2830772.2830785

[27] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the Front-End Bottleneck with Shotgun." ACM, 2018, pp. 30–42.

[28] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A Metadata-Free Architecture for Control Flow Delivery." IEEE Computer Society, 2017, pp. 493–504.

[29] E. Lab. (2022) vswarm-u: Microarchitecture for serverless workloads. [Online]. Available: https://github.com/ease-lab/vSwarm-u

[30] C. Lam, "Hot chips 2024: Qualcomm's oryon core," *Chips and Cheese*, August 2024. [Online]. Available: https://chipsandcheese.com/p/hot-chips-2024-qualcomms-oryon-core

[31] C. Lam, "Intel's redwood cove: Baby steps are still steps," *Chips and Cheese*, September 2024. [Online]. Available: https://chipsandcheese.com/p/intels-redwood-cove-baby-steps-are-still-steps

[32] C. Lam, "Lion cove: Intel's p-core roars," *Chips and Cheese*, September 2024. [Online]. Available: https://chipsandcheese.com/p/lion-cove-intels-p-core-roars

[33] C. Lam, "Arm's cortex x925: Reaching desktop performance," *Chips and Cheese*, March 2026. [Online]. Available: https://chipsandcheese.com/p/arms-cortex-x925-reaching-desktop

[34] O. Lempe. (2024) Next gen p-core: The lion cove architecture. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/824430/2024-intel-tech-tour-next-gen-p-core-the-lion-cove-microarchitecture.html

[35] Y. Liu, X. Li, T. Zhang, T. Liu, Q. Guo, F. Zhang, and J. Wang, "Avm-btb: Adaptive and virtualized multi-level branch target buffer," *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 17–31, 2024.

[36] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, "The gem5 Simulator: Version 20.0+," *CoRR*, vol. abs/2007.03152, 2020.

[37] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.

[38] K. Nesbit and J. Smith, "Data Cache Prefetching Using a Global History Buffer," pp. 96–96, 2004.

[39] P. Pathak, M. Sarwar, and S. Sohoni, "Markov prediction scheme for cache prefetching," in *Proceedings of 2nd Annual Conference on Theoretical and Applied Computer Science, November 2010, Stillwater, OK*. Association for Computing Machinery, 2010.

[40] A. Perais and R. Sheikh, "Branch target buffer organizations," *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 240–253, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:266085503

[41] G. Reinman, B. Calder, and T. M. Austin, "Fetch Directed Instruction Prefetching." ACM/IEEE Computer Society, 1999, pp. 16–27.

[42] D. Schall. (2023) Fetch directed instruction prefetching for gem5. [Online]. Available: https://github.com/dhschall/gem5-fdp

[43] D. Schall. (2025) Speculative update for tage-sc-l. [Online]. Available: https://github.com/gem5/gem5/pull/1854

[44] D. Schall. (2026) Implement ittage. [Online]. Available: https://github.com/gem5/gem5/pull/2906

[45] D. Schall, M. Ďuračková, and B. Grot, "The last-level branch predictor revisited," *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 1–16, 2026. [Online]. Available: https://api.semanticscholar.org/CorpusID:286242777

[46] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: Characterization and optimization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 757–770.

[47] D. Schall, A. Sandberg, and B. Grot, "Warming up a cold front-end with ignite," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 2023, pp. 254–267. [Online]. Available: https://doi.org/10.1145/3613424.3614258

[48] D. Schall, A. Sandberg, and B. Grot, "The last-level branch predictor," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '24. IEEE, 2024.

[49] A. Seznec, "A 64-kbytes ittage indirect branch predictor," in *2nd JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction (CBP-2)*, 2011.

[50] A. Seznec, "Tage-sc-l branch predictors," in *Proceedings of the 4th Championship Branch Prediction*, 2014.

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

[52] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-block ahead branch predictors," in *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996*, B. Dally and S. J. Eggers, Eds. ACM Press, 1996, pp. 116–127. [Online]. Available: https://doi.org/10.1145/237090.237169

[53] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: Association for Computing Machinery, 2002, p. 45–57. [Online]. Available: https://doi.org/10.1145/605397.605403

[54] R. Suite. (2024) Renaissance suite: A modern benchmark suite for the jvm. [Online]. Available: https://renaissance.dev/

[55] P. Vahdatniya, J. Humecki, H. Kao, T. Li, A. Sedaghati, F. Su, R. Zhou, A. Bi, R. Azimi, and M. Goudarzi, "Deer: Deep runahead for instruction prefetching on modern mobile workloads," *ArXiv*, vol. abs/2504.20387, 2025. [Online]. Available: https://api.semanticscholar.org/CorpusID:278171228

[56] G. Vavouliotis, L. Alvarez, B. Grot, D. A. Jiménez, and M. Casas, "Morrigan: A composite instruction tlb prefetcher," *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:239011660

[57] G. Vavouliotis, T. Rollet, D. B. Bartolini, B. Grot, L. Peled, and L. Yang, "Bumper: Hinting instruction usefulness for robust unified caches," in *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*, 2026, pp. 2029–2045.