

# The Last-Level Branch Predictor Revisited

David Schall 

Technical University of Munich  
Munich, Germany  
david.schall@tum.de

Mária Ďuračková 

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

Boris Grot 

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

**Abstract**—Branch prediction is critical for high-performance CPUs, with mispredictions causing significant execution inefficiencies. Modern server workloads exacerbate the challenge due to expanding instruction and branch working sets, while predictor capacities remain limited to avoid latency increases. A recently introduced hierarchical branch predictor design, LLBP, demonstrated a way to reduce misprediction rates by augmenting an unmodified TAGE-based predictor with a decoupled high-capacity metadata store. Despite using a large amount of storage, LLBP was shown to achieve only a fraction of the accuracy gain of an equal-sized (but impractical) TAGE-based predictor.

This work provides a detailed analysis of LLBP, identifying sources of its accuracy loss. Chief among these are contention within certain sets of LLBP’s high-capacity metadata store (namely those containing patterns for hard-to-predict branches), as well as duplication of patterns, which leads to prolonged training time. To address these issues, we propose *dynamic context depth adaptation*, an enhancement to the original LLBP design, which yields a significantly better distribution of patterns for hard-to-predict branches, thereby reducing both pattern set contention and pattern duplication. Our proposed design that realizes dynamic context depth adaptation requires only small modifications to the baseline LLBP while increasing its accuracy by 0.8–11.5% (average 3.6%).

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

## I. INTRODUCTION

Branch prediction is a key microarchitectural mechanism for today’s high-performance CPUs. The branch predictor keeps the pipeline fed with instructions by anticipating branch instructions and their outcomes. Each misprediction triggers a pipeline flush, rendering the work of tens of cycles and hundreds of instructions useless.

Modern server workloads, featuring massive instruction workings sets with complex control flow, pose a particular challenge for today’s branch predictors. A recent Google study reveals that branch mispredictions waste an average of 15.4% of execution cycles across thousands of their datacenter CPUs, resulting in millions of dollars in cost and energy losses [2].

Two factors are compounding the problem of branch predictor accuracy on server workloads. The first is the fact that instruction working sets (and, correspondingly, branch working sets) in server workloads are quickly growing in size. For example, Meta reports a 5.6x growth in the binary size of a major web service over four years [28]. The second is that branch predictor capacities are largely stagnant due to the fact that the branch predictor is a latency-critical structure, and enlarging its capacity would necessarily make it slower.

More accurate branch prediction would not only enhance the performance on today’s workloads but also enable more aggressive future CPU designs with wider pipelines and larger instruction windows [26]. Indeed, a large instruction window is useless if it cannot be kept full due to misspeculation-induced flushes. Our analysis of typical server workloads shows that the state-of-the-art 64KB TAGE-SC-L predictor [42] yields misprediction rates of 0.29–6.4 MPKI (avg. 2.91), causing a flush every 344 instructions on average. For comparison, an Intel Sapphire Rapids server CPU has a 512-entry ROB [38], which is impossible to fill given such misprediction rate.

Addressing the branch predictor accuracy challenge through larger branch prediction structures is an attractive approach, particularly because on-chip transistor budgets have rapidly expanded despite the slowdown in Moore’s law. Alas, as noted above, the branch predictor sits in the critical path to feed the pipeline with instructions, and any increase in predictor structures inevitably leads to higher prediction latency. And while a larger branch predictor has a positive effect on prediction accuracy, the increased accuracy gain is nullified by the longer access latency [20], [44].

Recent work has managed to break the capacity-latency trade-off in branch predictor design through a hierarchical organization, called LLBP, which combines an unmodified TAGE-based predictor in the first level with a high-capacity metadata store for TAGE patterns in the second level [37]. Crucially, the second level is entirely decoupled from the first, and is never accessed on the critical path of a prediction; instead, metadata (i.e., TAGE patterns) for upcoming branches are prefetched into a small buffer that is looked up in parallel with the primary TAGE using the same partial pattern-matching algorithm across the two structures.

LLBP solves two challenges for hierarchical TAGE-based predictors. First, TAGE’s aggressive hashing of branch addresses with global history annihilates the spatial locality of patterns belonging to a given branch. By scattering the patterns across the predictor’s storage space the hashing makes it difficult to identify which patterns belong to a given branch. Second, the massive variance in the number of patterns stored per branch; while most branches need only a few patterns, certain *hard-to-predict (H2P)* branches require hundreds to thousands. Analysis of Google datacenter traces [11] shows the most difficult branches have over 9K patterns each. Prefetching such a large number of patterns would require moving tens of KBs of metadata, which is clearly impractical.

LLBP addresses both challenges by localizing a small set of patterns, necessary for predicting an upcoming instance of a branch, in a *pattern set* of LLBP. Localization happens through the notion of a context, created by hashing a sequence of branch PCs leading up to the execution of the branch in question. For H2P branches with a large number of patterns, contexts naturally spread out the majority of patterns while keeping together the subset necessary for predicting a given branch instance. LLBP identifies upcoming contexts and prefetches only the subset of patterns needed for predicting a given context into a small in-core buffer.

While LLBP represents a breakthrough at a conceptual level, the original work’s design achieved less than a third of the opportunity of an equal-sized (but impractical) TAGE predictor. The objective of our work is to understand and address the deficiencies of LLBP’s design and, thereby improving LLBP’s accuracy at a given storage budget.

We analyze LLBP in detail and identify the main reasons for its poor accuracy as compared to TAGE. Our findings indicate that the limited number of patterns in a pattern set is a significant problem that affects only a small number of (H2P) branches but causes a large accuracy loss. Another problem is the fact that contextualization causes some easy-to-predict branches, requiring only short history lengths to predict, to be duplicated in multiple contexts. The duplication increases the training and adaptation time when branch behavior changes, since the patterns used for predicting these branches must be trained in multiple contexts, hurting the accuracy of LLBP.

We address both problems through a dynamic context depth adaptation approach that chooses the number of contexts to use for a given branch. For H2P branches requiring a large number of patterns, our approach uses a large number of contexts to maximally spread out the patterns to avoid thrashing inside the pattern sets. For easy-to-predict branches, our approach does the opposite and uses only a few contexts to minimize the redundancy and the training time. Moreover, dynamic context depth adaptation allows us to improve aspects of LLBP’s design that were constrained by its fixed choice of contexts.

Using a broad range of representative server workloads, we show that our proposed design, LLBP-X, improves prediction accuracy by 0.8–11.5% (average 3.6%) over the original LLBP at a similar storage budget through dynamic context depth adaptation and complementary enhancements.

Our contributions can be summarized as follows:

- We analyze LLBP and find the principal culprits behind its modest accuracy. Key among these are contention in some of the pattern sets (particularly those used by H2P branches) and the long training time due to redundancy induced by contextualization.
- We introduce dynamic context depth adaptation, an approach that chooses the number of contexts to use for a given branch. For H2P branches, dynamic context depth adaptation reduces contention inside pattern sets by spreading the patterns out across more contexts. For other branches, it minimizes the number of contexts used, thus

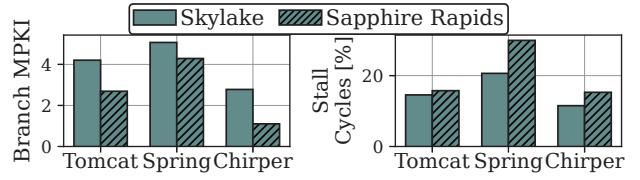


Fig. 1: Comparison of stall cycles due to conditional branch mispredictions and branch MPKI on Intel Skylake (solid) and Intel Sapphire Rapids (striped)

reducing both metadata redundancy and training time.

- We present LLBP-X<sup>1</sup>, an enhanced version of LLBP that addresses the identified bottlenecks. It improves accuracy by 0.8–11.5% (average 3.6%) at a similar storage budget.

## II. BACKGROUND

### A. Branch Prediction Bottleneck

Branch prediction plays a crucial role in modern high-performance CPU design. By anticipating branch instructions and their outcomes, the branch predictor ensures the processor pipeline remains fed with instructions. Each misprediction forces the CPU to flush its pipeline, wasting dozens of cycles and hundreds of already executed instructions. More accurate branch prediction not only enhances performance on today’s workloads but also enables more aggressive future CPU designs with wider pipelines and larger instruction windows [26].

To understand the importance of branch prediction in aggressive microarchitectures, we evaluate typical server workloads on an Intel Skylake CPU and a more recent Intel Sapphire Rapids CPU [38]. Compared to Skylake, Sapphire Rapids sports a much more aggressive microarchitecture including a much larger (>2x) ROB, wider pipeline, larger BTB, TLBs and caches, etc. We collect CPU performance counters while running these workloads and use Top-Down analysis to analyze the branch prediction bottleneck [52]. Refer to Section VI for methodological details.

First, we observe that Sapphire Rapids delivers a 40–52% (avg. 46%) lower CPI compared to Skylake. In Figure 1, we analyze the branch mispredictions per kilo instructions (MPKI) (left) and fraction of stall cycles due to branch mispredictions (right) for three workloads on Skylake (solid) and Sapphire Rapids (striped). While Sapphire Rapids achieves 15–60% (avg. 33%) fewer branch mispredictions compared to Skylake, the fraction of stall cycles due to branch mispredictions increases by 7–45% (avg. 30%).

This rise in the fraction of stall cycles due to branch mispredictions despite a lower rate of mispredictions clearly points to the fact that while more aggressive CPU designs can reduce or eliminate many sources of stalls, branch mispredictions fundamentally carry a high performance overhead that cannot be masked. Hence, reducing branch MPKI is a paramount challenge for future CPU microarchitectures.

<sup>1</sup>Implementations of LLBP-X, including a gem5 model, are available at <https://github.com/dhschall/LLBP-X>

### B. The TAGE Branch Predictor

TAGe GEometric history length (TAGE) [29], [39]–[42], [46] is widely considered the state-of-the-art branch predictor [26] and has found broad adoption in commercial processors [1], [19], [31]. Based on *prediction by partial pattern matching* (PPM) [29], TAGE correlates current branch behavior with previously observed control flow patterns using multiple predictor tables with geometrically increasing global history lengths. Modern implementations utilize up to 30 tables with history lengths reaching 3000 bits [42]. Each table entry contains a *tag*, a signed *prediction* counter for direction, and a *useful* bit for replacement decisions. A hash of the branch address and global history generates table indices and tags for pattern matching.

The longest matching history pattern serves as the *Prediction provider*, with a bimodal table (BIM) serving as fallback. On mispredictions, TAGE successively creates patterns with longer histories. In its latest form, TAGE-SC-L (referred to as TSL in the remainder of this paper), adds a *statistical corrector* for statistically biased branches and a *loop exit predictor*.

**The Capacity-Latency Tradeoff:** Larger branch predictors generally achieve higher accuracy, particularly for server workloads with their large branch working sets and complex control flow patterns. These workloads require tracking many instances of thousands of branches, overwhelming predictor structures. For example, increasing the capacity of TAGE-SC-L from 64KiB to 512KiB reduces mispredictions by 12.7–46.1% (avg. 27.5%) [20], [22], [26], [37]. However, the predictor sits on the critical path of instruction delivery, making latency a crucial constraint. Modern processors fetch and decode 8–10 instructions in parallel, often containing multiple branches per fetch bundle. Even a small increase in branch prediction and redirection latency can substantially impact performance, as it disrupts the steady flow of instructions into the pipeline. Research has shown that beyond a certain capacity threshold, the performance degradation from increased prediction latency outweighs the benefits of improved accuracy [20].

### C. The Last-Level Branch Predictor

To overcome the trade-off between storage and latency, recent work has proposed a hierarchical branch predictor organization called Last-Level Branch Predictor (LLBP) [37]. LLBP extends the TAGE-based predictor in the first level with a high-capacity second-level metadata store, effectively decoupling the need for high-capacity storage and fast prediction. The key insight behind LLBP is that branch predictor metadata exhibit context locality, i.e., the set of patterns necessary to predict a branch is determined by the global control flow (e.g. function call chain leading up to the execution of a branch).

LLBP leverages context locality to decompose the branch working set and associate each TAGE pattern with a specific program context. Specifically, LLBP uses a hash of several unconditional branches (UBs) preceding a conditional branch to determine the global control flow path leading up to that branch and form a context. All patterns capturing correlations

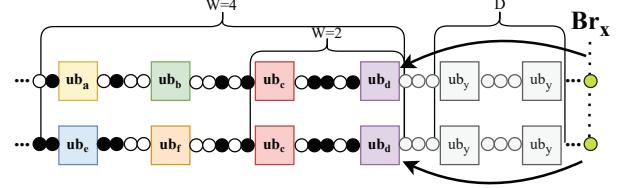


Fig. 2: Forming a context with different context depth (W). Circles represent taken/not taken conditional branches, and squares the unconditional branches used in context formation.

on the same global path (i.e., the same sequence of UBs) form a *pattern set* needed to predict the branch in this context.

Pattern sets are stored in a high-capacity metadata store (i.e., the LLBP pattern store). To overcome the longer access latencies of larger structures, LLBP uses global context information to precisely identify pattern sets required for upcoming contexts and prefetches these pattern sets ahead of time into a *pattern buffer* – a small in-core structure enabling low-latency predictions. The pattern buffer operates alongside the baseline TAGE-based predictor and uses the same partial-pattern matching algorithm as TAGE, logically extending TAGE to accommodate the prefetched patterns.

**C.1) Context-based Organization** LLBP’s context-based organization addresses a fundamental challenge in branch prediction: the high variance in required patterns per branch [37]. While most branches need a few patterns, some hard-to-predict branches require thousands. TAGE-SC-L handles this through aggressive hashing and bank interleaving [42], distributing patterns across all tables. Although this enables high storage density, it limits scalability as all tables must be accessed for every prediction. In contrast, LLBP bundles patterns into fixed-size sets of 16 patterns per context, enabling efficient prefetching but potentially trading off TAGE’s storage efficiency – contexts requiring fewer patterns waste capacity while those needing more patterns sacrifice coverage.

**C.2) Context Formation** The key parameters in forming a context hash are  $D$  and  $W$ .  $D$  refers to the number of most recent UBs deliberately ignored when creating the context ID hash. Doing so provides a temporal window during which LLBP’s access latency is hidden. The context depth  $W$  refers to the actual number of UBs (preceding the  $D$  skipped UBs) used to form the context hash. The original LLBP design used  $D = 4$  and  $W = 8$ , but these values can be adapted as needed.

Choosing the value for  $W$  involves a trade-off: a larger  $W$  provides more global context, enabling more precise pattern localization and reducing the number of patterns per context, but introduces redundancy for branches with short-range correlations. Figure 2 illustrates this trade-off with two executions of the same conditional branch  $Br_x$  through different history patterns. These paths share two recent unconditional branches  $ub_d$  and  $ub_c$ , but differ in preceding UBs.

Suppose a conditional branch  $Br_x$  requires a long history to make accurate predictions. With a small  $W$  (e.g.,  $W = 2$ ), both branch invocations would use only branches  $ub_d$  and  $ub_c$  to form a context ID, placing both of the patterns in the same

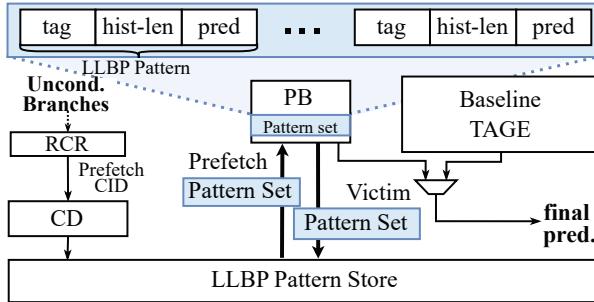


Fig. 3: LLBP design overview.

pattern set and increasing the risk of pattern set overflow. In contrast, a larger  $W$  (e.g.,  $W = 4$ ) would create distinct contexts for the two dynamic instances of the branch, spreading them across multiple pattern sets and reducing overflow potential.

Conversely, if the conditional branch  $Br_x$  is predictable with a short history pattern shared between invocations, a small  $W$  (e.g.,  $W = 2$ ) naturally hashes to the same context ID. Using a larger  $W$  would create different IDs, requiring duplication of the pattern across multiple pattern sets, causing redundancy.

### C.3) LLBP Design Overview

Figure 3 shows how LLBP supplements an existing 64K TSL branch predictor with four main components. The *LLBP pattern store* (PS) stores pattern sets for different contexts. The *pattern buffer* (PB) holds and implements prediction logic for the currently active context and caches recently accessed contexts. The *rolling context register* (RCR) maintains unconditional branch addresses to compute the current context ID using a rolling hash. The *context directory* (CD) enables associative metadata lookup for pattern sets using context IDs.

**Making Predictions:** LLBP uses TAGE’s partial pattern-matching algorithm [29] but a different pattern organization. While TAGE dynamically allocates storage to patterns across multiple tables, LLBP maintains a single pattern set per context. Each pattern consists of a prediction counter, pattern tag, and history length field, which determines how much global history is used for tag computation.

The Pattern Buffer (PB) implements a prediction logic similar to that of TAGE by performing parallel tag matches for all patterns in the current context’s pattern set. Upon a match, the sign of the prediction counter determines the predicted direction. If no tag matches, LLBP does not provide a prediction for this cycle. To arbitrate multiple tag matches, LLBP selects the longest matching pattern. The final selection between LLBP and the baseline predictor (TAGE) is based on the history length of the providing pattern where LLBP overrides the baseline prediction only if it matches on a pattern with the same or longer history length than TAGE.

**Prefetching Pattern Sets:** Accessing pattern sets from LLBP requires multiple sequential operations: computing a context ID, checking the CD, and reading from LLBP storage if the pattern set isn’t already cached in the PB. To hide this multi-cycle latency without introducing complex prediction mechanisms for future contexts, LLBP employs a prefetching

mechanism that associates the current context’s pattern set with a past context ID. Consequently, the current context ID triggers prefetches for upcoming pattern sets.

**Learning Pattern Sets:** Once a branch is resolved, the pattern (TAGE or LLBP) is updated based on the actual outcome. Upon misprediction, both TAGE and LLBP allocate a pattern with a longer history than the incorrect pattern. LLBP’s allocation process consists of three steps. First, if, for the current context, no pattern set exists, LLBP creates a new pattern set in the PB and its context ID is written to the CD. The replacement policy favors sets with more high-confidence patterns. Second, within a pattern set, LLBP replaces the least-confident pattern, setting the prediction counter to low confidence, taken or not taken.

**Writebacks:** A pattern set remains in PB during instruction execution and is updated at commit time. Modified, valid pattern sets are written back to the pattern store upon eviction.

**C.4) Design Tweaks** To make LLBP feasible for hardware implementation, modifications were necessary to reduce complexity while maintaining prediction accuracy. When making a prediction, LLBP selects the pattern with the longest branch history from the pattern set. To enable this, LLBP keeps the patterns in a pattern set sorted by history length. To reduce hardware complexity, LLBP organizes the 16 patterns into four buckets of four patterns each, with each bucket covering a specific history length range, limiting sorting to four patterns per bucket instead of all 16. As a further simplification, LLBP keeps only the 16 most commonly occurring history lengths (4 history lengths per bucket) out of TAGE’s original 21. Lastly, if LLBP provides the prediction (by having a longer pattern matching than the baseline TSL), the Statistical Corrector (SC) is suppressed.

**C.5) LLBP Prediction Accuracy** We evaluate the efficacy of LLBP over a 64K TSL [42] baseline on a set of 14 representative server workloads. We compare LLBP against an idealized 512K TSL version, which has a 0-cycle access latency and about the same storage budget as LLBP<sup>2</sup>. While the 512K TSL is unrealizable due to the long access latencies associated with the large predictor structures, it serves as an upper bound for the accuracy improvement achievable with 512KB of predictor capacity. To quantify the potential headroom beyond 512KB capacity, we also evaluate an infinitely-sized TSL. Finally, we consider a 0-cycle access latency version of LLBP (LLBP-0Lat) to understand the potential accuracy improvement if the latency constraints were lifted.

Figure 4 compares the branch mispredictions of LLBP and 512K TSL, normalized to a 64K TSL baseline that exhibits an absolute MPKI of 0.26-5.38 (avg. 2.92) (not shown in the graph but listed in Table I). We find that LLBP achieves a reduction in branch mispredictions of 0.6-25% (avg. 8.8%) over the baseline 64K TSL, corroborating prior work [37]. However, the accuracy improvement of LLBP is still far from the idealized 512K TSL, which achieves a reduction of

<sup>2</sup>The LLBP design can store 224K patterns in the LLBP pattern store plus 30K patterns in the baseline 64K TSL. 512K TSL can hold 240K patterns.

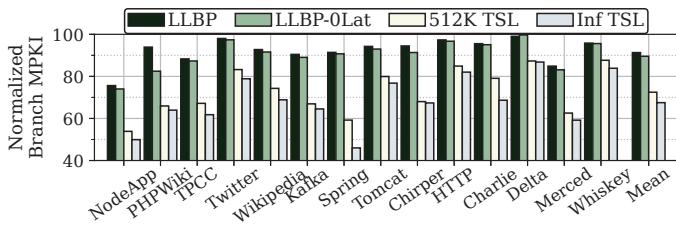


Fig. 4: Branch MPKI of LLBP, 512K TSL, and Inf TSL normalized to 64K TSL. The absolute MPKI of 64K TSL is 0.26-5.38 (avg. 2.92) and listed for each workload in Table I.

12.7-46.1% (avg. 27.5%) over the baseline. Even the 0-cycle access latency version of LLBP (LLBP-0Lat) falls short of the idealized 512K TSL, revealing a significant inefficiency of LLBP, given that it can store roughly the same number of patterns as 512K TSL. Finally, we observe that an infinitely-sized TSL reduces mispredictions by 13.2-54% (avg. 32.5%) over the baseline, representing only a modest improvement of 6.8%, on average, over the 512K TSL, thus indicating that 512KB capacity captures most of the available opportunity.

### III. IDENTIFYING LLBP’S ACCURACY BOTTLENECKS

In this section, we perform a comprehensive analysis of LLBP to understand how its design choices affect prediction accuracy. Our focus is on identifying bottlenecks that prevent LLBP from matching TAGE at the same storage budget.

#### A. LLBP in the Limit

We begin our analysis with a detailed limit study of LLBP, investigating the effect of various parameters and microarchitectural constraints. We examine both TAGE and LLBP with a zero access latency so as to focus the analysis exclusively on accuracy (measured in MPKI) and how it is affected by various parameters. Our metric of interest is MPKI reduction relative to the 515KB 0-latency LLBP design as presented in [37].

Figure 5 presents a stepwise analysis where we progressively remove LLBP’s various design constraints. We first remove LLBP’s design tweaks meant to improve its practicality given latency and storage constraints (+ No Design Tweaks). There are three such tweaks: bucketing (which limits the number of patterns with a given history length in a pattern set), excluding less-common history lengths used by the primary TAGE (which limits the history lengths that LLBP can store), and disabling SC override when LLBP provides a useful prediction. Disabling these affords fully-associative pattern sets, accommodates all 21 history lengths used by the primary TAGE predictor, and re-introduces SC override.

Next, we increase the pattern tag size to 20 bits (from 13 bit) to match TAGE’s pattern entropy (+ 20b Tag)<sup>3</sup>. We then allow unlimited contexts (from 14K contexts in baseline LLBP) with 31-bit context tags to eliminate aliasing (+ Inf Contexts), followed by allowing an unlimited number of patterns per pattern

<sup>3</sup>TAGE uses a 10-bit index and 8-bit tags for short histories and 12-bit tags for long histories. For Inf TSL, we don’t increase the table size; instead, we provide each table with unlimited associativity and remove aliasing by tagging each entry in addition with its PC.

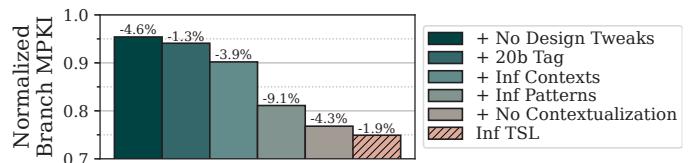


Fig. 5: Effect of successively removing LLBP’s design constraints on branch mispredictions normalized to the 0-latency LLBP baseline (lower is better). Percentages show the reduction relative to the preceding configuration.

set (+ Inf Patterns). Finally, we eliminate contextualization by replacing the RCR hash with the branch PC as context ID, effectively creating one context per branch with no limit on the number of patterns (+ No Contextualization).

Results of the study are shown in Figure 5, which plots MPKI improvement relative to LLBP-0Lat. Removing design tweaks yields a 4.6% MPKI reduction. Longer tags provide only a small 1.3% MPKI improvement. Unlimited contexts deliver a 3.9% MPKI reduction, indicating that the total number of pattern sets in the 512KB LLBP design is not a significant limitation. The largest MPKI reduction is achieved by allowing an arbitrary number of patterns per set (9.1% improvement), indicating that conflicts with the pattern sets are a major source of accuracy loss. Disabling contextualization provides the third-largest gain of 4.3%, pointing to an overhead caused by patterns being replicated across multiple contexts.

With all constraints removed, we find that LLBP is able to nearly match the accuracy of an infinite-sized TAGE. The small remaining accuracy gap is directly attributed to the temporal window,  $D$ , which LLBP uses to initiate prefetches early enough so as to hide its access latency. With  $D$  set to 0, we find that LLBP is able to match the accuracy of TAGE (not shown in the figure).

Our main finding is that while multiple factors impact LLBP’s accuracy, the two design concepts of LLBP that affect accuracy the most are the limited number of patterns per pattern set and contextualization, together responsible for more than half of the accuracy gap against TAGE<sup>4</sup>. Thus, we shift our focus to these two issues.

#### B. Pattern Set Utilization

We first focus on the limited number of patterns per context, which is the single biggest source of LLBP’s accuracy shortfall as compared to TAGE. To better understand this issue, we examine the distribution of the number of *useful* patterns per context using the configuration with unlimited patterns and contexts (+ Inf Patterns Figure 5). A LLBP pattern is considered useful when its prediction correctly overrides the baseline 64K TSL, which would otherwise mispredict.

<sup>4</sup>We note that bottlenecks can overlap and may prevent achieving the full opportunity when removing individual bottlenecks. However, we found that when removing only a single bottleneck per experiment (and reinstating the bottleneck before testing the next), the same trends hold – the main bottlenecks continue to be the limited number of patterns per pattern set and contextualization.

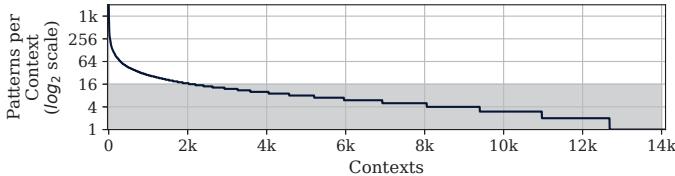


Fig. 6: Useful patterns ( $\log_2$  scale) per context for NodeApp. Contexts are sorted by the number of useful patterns. The shaded portion indicates LLBP’s limit of 16 patterns per pattern set and 14K contexts.

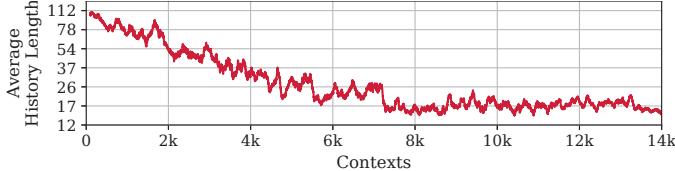


Fig. 7: Average history length of useful patterns per context for NodeApp. The contexts are sorted by the number of useful patterns they contain (same order as in Figure 6).

We present the results in Figure 6, where the number of useful patterns per context is sorted from the largest to the smallest. For clarity of exposition, we focus on just one application (NodeApp) and note that the trends hold for all of the evaluated workloads.

The figure reveals that the pattern distribution is highly skewed. The vast majority of contexts have sufficient capacity to accommodate all useful patterns (recall each pattern set, aka context, in LLBP contains 16 patterns). Only 14% of contexts exceed the pattern set capacity. Meanwhile, 68% of all contexts contain 8 useful patterns or fewer, pointing to a significant underutilization of LLBP’s pattern sets.

Referring back to Figure 5, which shows only a modest benefit from an infinite number of contexts, and combining that with the observation above that the majority of contexts are underutilized, we can conclude that there exists enough overall capacity in a 512KB LLBP to accommodate the useful patterns. However, *uneven distribution of patterns across contexts leads to poor space efficiency with some contexts experiencing high conflict rates while most contexts are underutilized*.

In order to understand why some contexts experience high contention when the majority does not, we hypothesize that the highly-contended contexts contain patterns for hard-to-predict branches. Prior work has shown that such branches result in many patterns (often numbering in hundreds or thousands) of long history length being allocated in TAGE [37], which would explain why the LLBP contexts containing the corresponding patterns would be overwhelmed.

To validate this intuition, we plot the average history length of useful patterns per context in Figure 7. In the figure, the contexts are sorted by the number of useful patterns they contain, which is the same order as in Figure 6. The figure confirms our hypothesis; the contexts containing the most useful patterns (left-hand side of the figure) are also the ones with patterns having the longest history (average history length

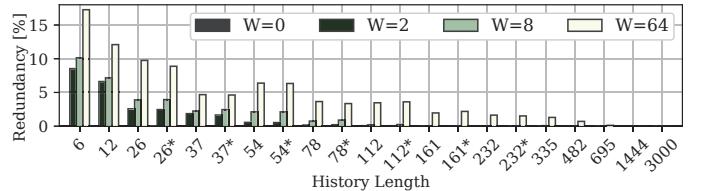


Fig. 8: Duplication of patterns as a function of TAGE history lengths and context depth  $W$  for NodeApp. The asterisks indicate histories that differ in hash function, not length [42].

up to 112). Meanwhile, contexts containing the fewest useful patterns (right-hand side of the figure) store patterns with the shortest history length (average length of 17).

### C. Understanding Contextualization

Figure 5 shows that contextualization has the second-largest impact on accuracy of LLBP. Here, we analyze contextualization and its impact on accuracy.

Recall from Section II-C that an LLBP context is identified through a hash of ( $W$ ) recently-executed unconditional branch PCs, with LLBP using  $W=8$ . The parameter  $W$  is fundamental to LLBP; larger  $W$  values create more specific contexts by considering longer unconditional branch history, while smaller  $W$  values yield less specific contexts. In the limit,  $W = 0$  lumps all patterns into a single context, whereas larger  $W$  values spread patterns across multiple contexts for better distribution.

While large values of  $W$  seem highly desirable for their ability to spread patterns, that comes at a cost, which is duplication of patterns, particularly those with a short history. Consider a conditional branch  $Br_x$ , requiring a very short history for prediction, and residing inside a function called through an unconditional branch  $ub_d$  (see Figure 2). The more unique contexts for  $ub_d$  exist in LLBP, the more times the pattern(s) needed for predicting  $Br_x$  would be replicated<sup>5</sup>.

In Figure 8, we quantify this behavior by showing the duplication of patterns in LLBP as a function of history length for different values of  $W$ . Here, duplication is defined as a total number of useful patterns for a given history length divided by the number of unique patterns for that history length. The history lengths are the ones used by the baseline TAGE.

The figure shows a clear trend, with patterns of short history length experiencing high degrees of duplication, which decreases as the history length goes up. The duplication is larger for bigger values of  $W$ . For instance, at history length 6 and  $W = 2$ , 8.5% of patterns are duplicates, which increases to 10.1% for  $W = 8$  and 17.2% for  $W = 64$ . At a longer history length of 78, 0.2%, 0.9% and 3.3% of patterns are duplicates for  $W = 2, 8$  and 64, respectively.

*Pattern duplication in LLBP leads to three issues: wasted capacity; longer training time, as each context must learn*

<sup>5</sup>The reason for shorter patterns being duplicated more, particularly for larger  $W$ , is the inherent correlation between LLBP’s unconditional branch history (used for context formation) and TAGE’s all-branch history. As a larger  $W$  spans more conditional branches tracked by TAGE (Figure 2), a branch correlated only with recent history bits must replicate the same pattern in every context deeper than the required pattern length.

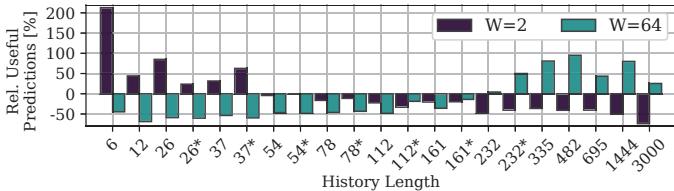


Fig. 9: Useful predictions made from different history lengths for various context depths  $W$  relative to  $W = 8$  for NodeApp.

patterns independently; and slower adaptation to behavioral changes, since updates occur separately. Each contributes to increased mispredictions.

#### D. Summary

There exists a 25.1% accuracy gap between a 0-latency 512KB LLBP and an infinite-capacity TSL. Over half of this gap (53%) is due to the limited capacity of the pattern sets and the overhead of contextualization. The capacity issue affects only a small fraction (15%) of contexts yet has a particularly acute impact on accuracy, the reason being that these contexts store patterns for hard-to-predict branches requiring a multitude of patterns. Contextualization is how LLBP spreads the patterns across contexts; while greater degrees of contextualization (higher  $W$ ) can do a better job of spreading the patterns and reduce capacity pressure in the pattern sets, it comes at a cost of pattern duplication which increases training and adaptation time, and wastes pattern set capacity.

### IV. LLBP ZEN THROUGH BETTER CONTEXTUALIZATION

Contextualization in LLBP presents a tension. On the one hand, it can be used to better spread the patterns, thus avoiding the conflicts in the highly-contended pattern sets (left side of Figure 6). On the other hand, it creates problems through pattern duplication, resulting in increased training and adaptation time, and wasted LLBP capacity. Even at a modest contextualization degree of  $W = 8$  used by LLBP, duplication-related overheads cause a 4.3% accuracy loss.

The key contribution of this work is in reconciling the tension related to contextualization, thus leveraging it for improved context efficiency while minimizing the overheads. Our key insight is that only a small fraction of the pattern sets (the ones dominated by patterns with long history length per Figure 7) suffer from contention; thus, only these pattern sets benefit from a larger  $W$  (which we refer to as a high *context depth*). Meanwhile, the vast majority of the pattern sets can enjoy a small  $W$  (shallow context depth), which would reduce duplication and the associated overheads particularly in training and adaptation time.

We validate this intuition through a limit study that assess the useful predictions for each history length as a function of context depth  $W$ . The results are plotted in Figure 9, which shows the change in useful predictions for  $W = 2$  and  $W = 64$  normalized to the LLBP baseline of  $W = 8$ .

As expected, for short pattern lengths (6-37 bits), a shallower context ( $W = 2$ ) dramatically improves efficiency by

reducing duplication and the associated overheads. For these pattern lengths, useful predictions increase by 63-213%. Conversely, high context depth ( $W = 64$ ) is detrimental with fewer useful predictions relative to the  $W = 2$  baseline. The trends are reversed for long pattern lengths (232-3000 bits), for which a high context depth helps spread out the patterns across many contexts, thereby increasing useful predictions by 4.2-95%, while a shallow context depth decreases useful predictions by 49-74% relative to the  $W = 8$  LLBP baseline.

### V. LLBP-X

#### A. Overview

We introduce LLBP-X, an augmented LLBP design that addresses the main shortcomings of original LLBP: (1) the skewed distribution of patterns across contexts and (2) the redundancy for short patterns induced by contextualization.

LLBP-X is based on the insight that adapting the context depth can effectively overcome both shortcomings. Most contexts benefit from a shallow context depth, which increases pattern set utilization and reduces redundancy. Conversely, a few contexts, particularly those containing hard-to-predict branches that require many patterns, benefit from a deeper context depth. By distributing the patterns of these contexts across more pattern sets, a deep context can achieve higher coverage and reduce potential conflicts.

LLBP-X introduces a dynamic approach to context depth adaptation, addressing the inherent trade-off between storage efficiency and pattern coverage. By default, context depth is set to  $W = 2$  to minimize redundancy and training time. When a context accumulates a significant number of confident patterns, LLBP-X increases the context depth to  $W = 64$  for that specific context, effectively spreading patterns across additional pattern sets, minimizing conflicts. LLBP-X uses only two  $W$  values as empirical studies showed only marginal accuracy gains with additional values, not justifying increased hardware complexity.

To determine when to increase context depth, LLBP-X employs two complementary heuristics: The first heuristic is based on the number of patterns in a context. When the number of patterns in one of the pattern sets exceeds a threshold  $T_{\max}$ , LLBP-X starts tracking the length of allocated patterns in the context. The second heuristic is based on the insight from Section III-B that the number of patterns correlates with the history length. LLBP-X increases the context depth to  $W = 64$  when the average history length of the patterns in the context exceeds a threshold  $H_{\text{th}}$ .

To further improve coverage, LLBP-X couples the dynamic context depth adaptation with history length selection. Recall that the original LLBP design limited the set of history lengths to just 16 out of the 21 maintained by TAGE, thus sacrificing coverage in order to keep the design simple and fast. Based on the observation in Section IV, we note that the choice of context depth naturally correlates with history length – shallower contexts are chosen because most patterns have a short history length, whereas deeper contexts must keep longer history lengths. Based on this insight, LLBP-X can

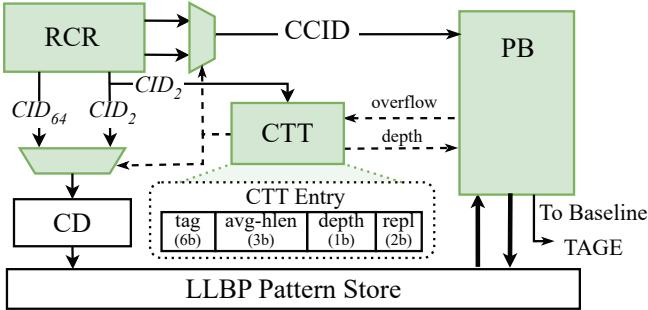


Fig. 10: LLBP-X architecture overview. Green are the components modified from the original LLBP design.

accommodate all 21 history lengths while keeping hardware fast and simple by allowing deeper contexts to keep longer histories, and shallow contexts shorter ones.

Figure 10 provides an overview of LLBP-X’s architecture, where the green components represent the newly introduced components and modifications to the baseline LLBP design.

### B. Dynamic Context Depth Adaptation

**B.1) Learning Context Depth** A key objective of LLBP-X is to detect contexts suffering from high contention, which would therefore benefit from increased context depth. By default, every context uses a shallow context depth ( $W = 2$ ) to maximize storage efficiency and improve training time. LLBP-X dynamically responds to high pressure in pattern sets by transitioning to a deeper context depth of  $W = 64$ , effectively providing additional capacity for contexts with many patterns.

To achieve this, LLBP-X utilizes an additional set-associative structure, the *Context Tracking Table* (CTT), which monitors highly contended contexts and serves as a selector determining whether a context should use shallow or deep context depth.

LLBP-X identifies contended contexts using the PB (Section II-C) to monitor pattern set utilization through the confidence bits of patterns’ prediction counters. When the number of confident patterns in a set exceeds a predefined threshold, the PB signals the CTT via an overflow signal to begin tracking that context.

Once tracked in the CTT, the transition to deeper context is not instantaneous. Instead, the CTT monitors the history length of pattern allocations using a saturating counter (*avg-hist-len*). When a pattern is allocated that exceeds a threshold history length ( $H_{th}$ ), *avg-hist-len* increments; otherwise, it decrements. Upon the *avg-hist-len* counter reaching saturation, the context transitions to deep context ( $W = 64$ ) by setting the *depth* bit. The rationale behind the history-based adaptation is based on our insight that the length of a history and the depth of contexts are tightly correlated: longer patterns typically arise in contexts with complex branching behavior, which benefit from deeper context.

Figure 10 visualizes the CTT entry organization, consisting of a tag, the *avg-hist-len* counter, a *depth* bit indicating shallow ( $W = 2$ ) or deep context depth ( $W = 64$ ), and replacement

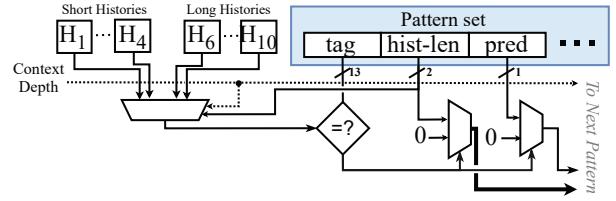


Fig. 11: LLBP-X’s pattern matching with history length range selection for the first pattern in a pattern set

bits. Each CTT entry maps to exactly one shallow context ( $W = 2$ ) regardless of the actual context depth used. For  $W = 2$ , the CTT entry naturally corresponds to a single CD entry. At  $W = 64$ , one CTT entry effectively tracks multiple CD entries.

The adaptive mechanism remains active even after switching to  $W = 64$ . If branch behaviour changes and the history length of allocated patterns becomes shorter, the context can dynamically revert to the shallow  $W = 2$ , ensuring continuous use of the optimal context depth. The switching threshold adds a hysteresis to avoid a ping-pong effect. Nevertheless, each transition incurs a cost: patterns from the previous depth are lost and must be relearned from scratch. This switching penalty is the main reason that more than two distinct context depths don’t lead to additional performance gains—the retraining overhead offsets the gains from finer adaptation granularity.

**B.2) Generating Context ID** The Rolling Context Register (RCR) is modified to compute two distinct rolling context IDs:  $CID_2$  and  $CID_{64}$ .  $CID_2$  is hashed from two unconditional branches, creating a context ID for a shallow context depth ( $W = 2$ ). Likewise,  $CID_{64}$  is computed using a hash of 64 unconditional branches, representing a deep context.

Upon predicting an unconditional branch, the RCR simultaneously updates both context IDs. These IDs are then routed through a multiplexer controlled by the CTT. The selection mechanism works as follows:  $CID_2$  is used to index the CTT, and upon a hit, the *depth* bit determines whether to select  $CID_2$  or  $CID_{64}$ . Upon a tag miss  $CID_2$  is selected.

The selected context ID is subsequently used to access the context directory (CD), which proceeds with the standard prefetch operation if a pattern set for the context is found.

The selection between  $CID_2$  and  $CID_{64}$  is required for both the prefetch context ID (PCID) and the current context ID (CCID, which indexes the Pattern Buffer). While this suggests a need for two CTT accesses per RCR update, in practice, however, only one access is required for the PCID, as the depth bit can be cached between the prefetch trigger and the context activation in the Pattern Buffer (PB).

Crucially, the entire context depth selection process happens off the critical prediction path and does not interfere with the prediction logic of the pattern buffer.

### C. History Range Selection

The dynamic adaptation mechanism of LLBP-X provides an opportunity to re-examine and refine design choices in

the original LLBP [37]. One such design decision is the selection of history length. Recall, that the original LLBP design considered only a subset of 16 of the TAGE’s original 21 histories.

As our earlier investigations in Section IV reveal, contexts with a shallow depth ( $W = 2$ ) predominantly contain short patterns, whereas deep contexts ( $W = 64$ ) typically contain longer patterns. Exploiting this observation, LLBP-X uses two different ranges of history lengths based on context depth.

Under this refined strategy, patterns in shallow contexts ( $W = 2$ ) are restricted to use shorter histories (namely, the first 16 history lengths tracked by TAGE), while patterns in deep contexts ( $W = 64$ ) are restricted to TAGE’s 16 longer history lengths. If LLBP-X attempts to allocate a pattern with a history length outside the currently active range, the allocation is dropped. However, the *avg-hist-len* counter is still updated, which may eventually trigger a context depth transition. Our design continues to employ LLBP’s bucketing mechanism and splits the active histories evenly between four buckets.

By restricting the history lengths according to context depth, LLBP-X significantly improves pattern set utilization and reduces bucket conflicts, ultimately enhancing accuracy.

Figure 11 shows LLBP-X’s prediction path for the first pattern in a pattern set; the context depth bit serves a dual purpose: not only it indicates the context depth, but also feeds directly into the history length multiplexer. This bit effectively partitions the history length selection, allowing patterns in shallow contexts ( $W = 2$ ) to select between four short history lengths, while the patterns of deep contexts ( $W = 64$ ) choose between four long history lengths. The implementation requires minimal modification to the original LLBP design. Only the depth bit must be passed together with the CCID to the pattern buffer. Additionally, LLBP-X extends the LLBP’s four-way multiplexer to an eight-way multiplexer.

#### D. Discussions

**D.1) Multiple Predictions per Cycle** Today’s processors feature increasingly wider pipelines, requiring multiple branch predictions per cycle [7], [44]. LLBP-X is well-suited to perform multiple predictions per cycle: For conditional branches within the same context (with no intervening unconditional branches), a single PB access suffices by replicating the pattern-matching logic. For branches that cross different contexts, dual-porting the PB is needed to support simultaneous reads of two pattern sets. Similarly, fetching multiple unconditional branches per cycle requires extending the RCR, CD, and CTT to compute and look up multiple consecutive PCIDs per cycle.

**D.2) LLBP-X in an Overriding Predictor** Modern branch predictors cannot make a prediction in a single cycle due to high clock frequencies and architectural complexity [1], [21], [45]. Instead, they employ overriding schemes where a fast, simple predictor (e.g. bimodal) provides an initial prediction, later confirmed by the TAGE predictor [12], [31], [54]. When TAGE disagrees, the pipeline must be redirected, exposing TAGE’s prediction latency.

Application (branch MPKI)	Description
NodeApp (4.43)	NodeJS webserver
PHPWiki (3.08)	PHP wiki web server
TPCC (3.74), Twitter (3.03), Wikipedia (2.52)	Java BenchBase suite [10]
Kafka (0.26), Spring (3.58), Tomcat (3.4)	Java DaCapo suite [5]
Finagle-chirper (0.48), Finagle-HTTP (2.81)	Java Renaissance suite [48]
Charlie (2.89), Delta (1.09), Merced (4.13), Whiskey (5.38)	Google traces [11]

TABLE I: Workloads with branch MPKI for 64K TSL.

Core	4GHz, 8-way OoO, 576 ROB, 190/120 LQ/SQ
Branch Pred	64KiB TAGE-SC-L [42], LLBP [37], LLBP-X
BTB	16K entry, 8-way
Caches	L1-I: 64KiB, 16-way, 4 cycle, 10 MSHRs L1-D: 48KiB, 12-way, 5 cycle, 16 MSHRs L2: 3MiB, 16-way, 16 cycle, 32 MSHRs LLC: 8MiB, 16-way, 30 cycle, 64 MSHRs
Prefetchers	Instructions: FDIP, Data: BOP [30], L2: Next-line
Memory	DDR4 3200MHz, 12.5 ns RCD/RP/CAS

TABLE II: Parameters of the simulated processor.

LLBP-X offers two attractive design options in this context. First, its Pattern Buffer, the only component required for prediction, is significantly smaller than TAGE (< 5% in our implementation), enabling earlier overriding than TAGE and reducing redirection penalty (evaluated in Section VII-C). Second, LLBP-X could complement a smaller TAGE implementation, maintaining accuracy while reducing overriding penalties and ultimately improving the overall performance and energy efficiency. We touch on this trade-off in Section VII-G but save a more extensive evaluation for future work.

**D.3) Area Overhead** LLBP-X introduces three sources of storage overhead with respect to the original LLBP design. The newly added CTT contributes the largest overhead, adding 9KB of storage to the original LLBP (details in Section VI, sensitivity study in Section VII-F). Furthermore, the RCR depth is expanded from 8 to 64 unconditional branches, adding 224 bytes of capacity. Finally, we empirically found that a threshold of 7 confident patterns per pattern set before context tracking provides optimal performance. This necessitates increasing replacement bits in the CD from 2 to 3, adding 142 bytes. Correspondingly, LLBP-X incurs a total capacity overhead of 9.36KB, a 1.8% increase over LLBP.

## VI. METHODOLOGY

**Workloads** To facilitate a direct comparison with the original LLBP design, we study the same set of server traces used in that work [37] (see Table I). These comprise seven Java benchmark workloads [5], [10], [22], [48], two web services (NodeApp [13] and PHPWiki [17]), and four Google data center workloads [11]. All traces include both user and kernel space instructions, and are collected via gem5 [27].

**Hardware Experiments:** We use the following server setups to conduct hardware experiments in Section III-A:

**Sapphire Rappids:** A SuperServer SYS-121H-TNR [49] server featuring a 4th Gen. Intel Xeon 5418N Processor [18] and 128GB of DDR5 memory.

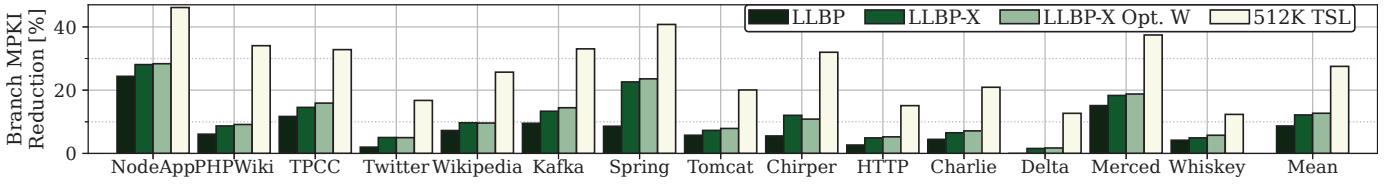


Fig. 12: Branch misprediction reduction over 64K TSL. Original MPKI shown in Table I.

**SkyLake:** A *c220g5* server node in the CloudLab cluster at University of Wisconsin [8], featuring an Intel Xeon Silver 4114 (dual socket 10-core) [51] and 192GB of DDR4 memory.

**Simulator Infrastructure:** For characterization and sensitivity studies focused on branch predictor accuracy, we use the trace-based simulation framework from LLBP [37], which enables rapid prototyping and design space exploration. For performance evaluations, we integrated LLBP-X into gem5 [4], [14], [27] v25.0.0.0, allowing detailed cycle-accurate, full-system simulations. We fixed the speculative history update of TAGE-SC-L in gem5 [34], integrated patches for the decoupled front-end [33], [36], and configured the system with parameters in Table II, reflecting a high-performance industry baseline [24], [25]. Branch predictor simulations execute 100M warmup and 200M measurement instructions, while gem5 simulations run for 200M warmup and 300M measurement for better cache warmup. Both the LLBP-X model and the gem5 port are publicly available to facilitate future research<sup>6</sup>. As the Google traces are only available in trace format and thus incompatible with gem5’s full-system simulation, we omit them in the performance evaluation.

#### Simulated designs:

**64K TSL:** The baseline in this work is 64KB TAGE-SC-L [42].

**LLBP:** as proposed in [37], augmenting a 64K TSL. 64-entry PB, 14K contexts, and 16 patterns per pattern set. Storage budget: 515KB. We use 16 histories, a prefetch distance  $D = 4$ , a context depth  $W = 8$ , and 6 cycles access latency for LLBP.

**LLBP-X:** Our proposal uses the baseline LLBP design as the starting point and makes the following modifications (parameters were determined using empirical studies):

The CTT is 6-way set associative with LRU replacement. Each entry contains a 6-bit tag, a 3-bit avg-hist-len counter, one depth bit, and two replacement bits. The total capacity of the CTT is 9KB, which allows for tracking 6K contexts.

The pattern buffer sends the overflow signal once a pattern set is filled with 7 high confidence patterns. The saturating avg-hist-len counter is incremented whenever the history length of a pattern allocation exceeds the length 232, otherwise, the counter is decremented. The avg-hist-len counter triggers a depth switch once exceeding a value of 7. LLBP-X uses history lengths 6-232 for the shallow context depth ( $W = 2$ ) and 37-3000 for the deep context ( $W = 64$ ). The combined PB and baseline TAGE results are fed into the SC.

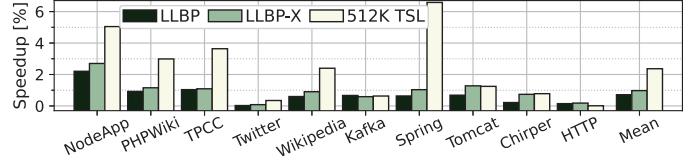


Fig. 13: LLBP-X’s speedup over 64K TSL

## VII. EVALUATION

### A. Prediction Accuracy

We first evaluate LLBP-X’s effectiveness in reducing branch mispredictions by comparing it against the original LLBP design [37]. To establish an upper bound for dynamic context depth adaptation, we include a configuration (called *LLBP-X Opt-W*) where the optimal context depth ( $W=2$  or  $W=64$ ) for each context is found ahead of time, thus avoiding the need to retrain when switching from a shallow to a deep context.

Figure 12 illustrates branch MPKI reduction across all 14 benchmarks relative to the 64K TSL baseline (absolute mispredictions listed in Table I). LLBP-X achieves consistent MPKI reductions of 1.4-27% (average 12.1%), peaking at 27% with NodeApp. This represents a 36% average improvement over the original design, with an absolute gain in branch prediction accuracy ranging from 0.8% to 11.5% (average 3.6%). Notably, these substantial accuracy improvements are achieved through minimal modifications to the baseline LLBP design and negligible additional area.

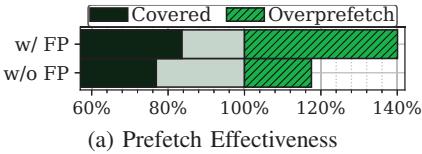
The optimal context depth configuration (LLBP-X Opt-W) reduces MPKI by 1.7-28.4% (avg. 12.6%), indicating that LLBP-X’s dynamic adaptation achieves accuracy within 97% of the optimal. For Chirper, LLBP-X manages to surpass the Opt-W configuration by continuously adapting  $W$  throughout execution. The combination of high efficiency, significant MPKI reduction, and minimal design complexity make LLBP-X a compelling option for adoption in real-world design.

Despite these gains, a substantial gap remains compared to an equally sized, idealized 512K TSL predictor, which achieves a 12.7-46.1% (avg. 27.5%) reduction in mispredictions. Closing this gap remains an open opportunity for future work, and the insights developed in this study provide a foundational understanding for further advancing LLBP’s contextualization mechanisms to reach TSL-like accuracy.

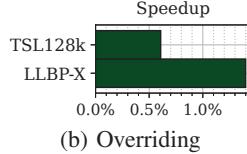
### B. Speedup

We use gem5 to evaluate LLBP-X’s performance improvement achieved through increased branch prediction accuracy by comparing its speedup over 64K TSL baseline with the

<sup>6</sup>CBP and gem5 models of LLBP-X: <https://github.com/dhschall/LLBP-X>



(a) Prefetch Effectiveness



(b) Overriding

Fig. 14: (a): Prefetch effectiveness with (upper) and without (lower) false path (FP) prefetchedes. (b): LLBP-X in an overriding scheme.

original LLBP design. For reference, we also include an idealized 512K TSL with 0-cycle access latency, representing the theoretical upper bound for comparable storage.

Figure 13 shows that LLBP-X achieves a 1% average speedup (0.08-2.7%) over the 64K TSL baseline, consistently outperforming LLBP’s 0.71% average speedup (0.02-2.2%). This represents 42% of the gains achieved by the ideal 512K TSL, which improves performance by 2.4% on average.

### C. Prefetch Efficiency, False Path and Overriding Effect

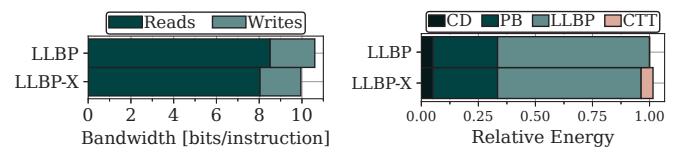
The execution-driven gem5 simulator enables detailed timing evaluation of LLBP-X’s integration with the core pipeline. We are interested in (1) the effectiveness of LLBP-X’s prefetching mechanism and (2) the effect of false path prefetchedes, and (3) LLBP-X’s integration with an overriding pipeline (discussed in Section D.2).

**Prefetch Efficiency:** We categorize prefetches into (1) prefetches that arrive too late in the PB, (2) those that arrive on time, and (3) prefetched pattern sets that are never used for prediction (no pattern matches). Our results in Figure 14a (upper bar) indicate high coverage, with 84% of prefetches arriving on time. However, only about two-thirds of the prefetches are useful for predictions (40% over-prefetches), revealing a significant opportunity for future work to reduce LLBP-X’s power consumption.

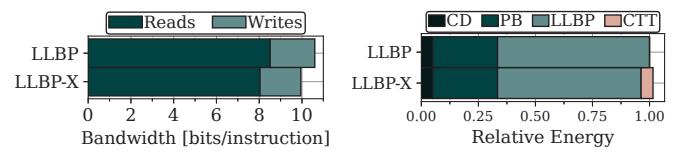
**False Path Effects:** To understand false path effects, we mark pattern sets with prefetch timestamps and flush pattern sets brought into the PB by false path instructions upon a misprediction. As shown in the bottom bar of Figure 14a, omitting false path prefetchedes reduces overprefetches by 56%, but also leads to a 8% reduction in coverage and a 1.4% drop in prediction accuracy (not shown in the graph), showing that false path prefetchedes provide benefit despite contributing to unused prefetches.

**Overriding Scheme:** We model an overriding scheme where the bimodal component of TAGE-SC-L and the PB of LLBP provide their prediction within one cycle. If the prediction from the TAGE or the SC components differs from and overrides this initial prediction, we model an overriding delay by stalling the decoupled branch predictor for 3 cycles, a duration based on open-source implementations of TAGE-SC-L [47], [54].

In Figure 14b, we compare the speedups of LLBP-X and a 128K TSL version relative to a 64K TSL baseline. All configurations utilize the same 3-cycle overriding scheme. The results show that the 128K TSL achieves a 0.6% average



(a) Transfer bandwidth



(b) Energy estimation

Fig. 15: Comparison of transfer bandwidth (a) and energy consumption (b) of LLBP-X and LLBP.

speedup, whereas LLBP-X achieves a 1.4% average speedup over the 64K TSL baseline.

### D. Transfer Bandwidth and Energy Estimation

In this section, we analyze LLBP-X’s communication bandwidth requirements and energy consumption characteristics.

**Bandwidth:** We evaluate the read and write traffic between LLBP-X’s pattern store and pattern buffer, comparing it with the original LLBP design. Both implementations transfer 288 bits per read or write transaction.

Figure 15a illustrates the bits transferred per instruction for both designs. Pattern set reads dominate the bandwidth consumption, with only a fifth on the pattern set writes. LLBP-X achieves a 6.1% reduction in transfer bandwidth compared to LLBP, requiring only 9.9 bits per instruction versus LLBP’s 10.6 bits, while simultaneously delivering higher prediction accuracy. The reason for LLBP-X having lower bandwidth usage is the reduced duplication of patterns due to shallow context and the more precise contextualization for deep contexts that reduces the volume of unused prefetches.

**Energy:** To assess energy efficiency, we compare LLBP-X’s energy consumption against the original LLBP design using CACTI 7.0 [3] for 22nm technology. The pre-existing structures—CD, PB, and LLBP—are modeled as 7-way, 4-way associative and directly mapped caches, respectively. The CD is 8-bit wide, whereas PB and LLBP are 36-byte wide. TAGE is modeled as a direct-mapped, 42-byte wide cache (21 tables \* (12b tag + 3b counter + 1b useful bit)). The new CTT component is modeled as a 6-way set associative structure with a 2-byte access width. Our analysis focuses only on the energy consumption of LLBP-X’s structures, excluding transfer energy and pipeline energy savings from improved prediction accuracy. The total energy consumption is calculated by weighting each structure’s access energy with its access frequency: the PB requires access every cycle, while CD and CTT are accessed only during unconditional branch execution, and LLBP-X’s pattern store is accessed exclusively on CD hits and pattern set writebacks.

Figure 15b shows LLBP-X’s energy consumption relative to the original LLBP. We observe that LLBP-X’s reduced volume of pattern set reads results in 5.4% lower access energy to the LLBP pattern store. However, the CTT (a new structure in LLBP-X) carries an additional energy cost of 5.2% relative to LLBP, which eats up these savings. As a result, LLBP-X increases energy consumption by 1.5% over the original LLBP.

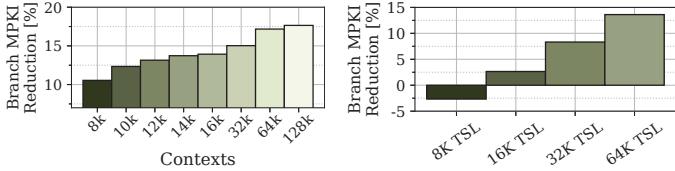


Fig. 16: LLBP-X pattern store size (a) and baseline TAGE size (b) sensitivity on MPKI reduction. (a) is relative to 64K TSL. (b) is relative to the corresponding baseline TSL. In both graphs, higher is better.

### E. Optimization Breakdown

LLBP-X introduces two key optimizations to enhance the accuracy of the baseline LLBP design: dynamic context depth adaptation and dynamic history range selection. To quantify the impact of each optimization, we analyzed their individual contributions to the overall accuracy improvement (graph omitted for brevity). We find, dynamic context depth adaptation accounts for 82% of the MPKI reduction compared to LLBP, while dynamic history range selection contributes 18%.

### F. Sensitivity Study

We perform a sensitivity study to determine the optimal parameters for  $H_{th}$  and the size of the CTT (the graphs for this study are omitted for brevity). For  $H_{th}$ , we sweep history length ranging from 37 to 1444 and measure the relative MPKI reduction compared to the 64 TSL baseline. We find the best performance is achieved with  $H_{th} = 232$  (13.6% reduction), and the worst performance was observed with  $H_{th}=1444$  (12.2% reduction). Most benchmarks show minimal sensitivity around the optimal value, with exceptions in Spring (optimal  $H_{th}=112$ ) and Merced (optimal  $H_{th}=1444$ ).

Similarly, we sweep the size of the CTT from 4K to 8K entries while maintaining a constant set size of 1K entries. We found that beyond 6K entries, no further reduction was observed, with 6K entries reducing mispredictions by 13.6% compared to 12.8% with 4K entries. Based on these findings, we configure LLBP-X with a 6K-entry, 6-way associative CTT, adding 9KB of storage (1.7% increase over LLBP).

### G. Sensitivity to Predictor Capacity

We investigate how LLBP-X’s pattern store capacity and baseline TAGE size affect prediction accuracy using a 0-cycle pattern store access latency model.

**LLBP:** We sweep the size of LLBP-X’s pattern store from 8K to 128K contexts (the baseline LLBP-X tracks 14K contexts), maintaining all other structures and using a fully associative context directory. Figure 16a shows LLBP-X’s MPKI reduction relative to the 64K TSL. The results demonstrate consistent accuracy improvements with increasing pattern store size, with MPKI reductions ranging from 10.5% (8K contexts) to 17.6% (128K contexts), indicating scalability with future transistor budgets.

**TSL:** We vary baseline TAGE sizes from 8K to 64K entries by adjusting pattern table entries (128 to 1K per table) while

maintaining the configuration of Statistical Corrector and loop predictor. Figure 16b shows that with a fixed 14K-context LLBP-X capacity, LLBP-X maintains effectiveness even with reduced TAGE sizes, achieving a 2.6% MPKI reduction with a 4x smaller baseline TAGE (16K TSL). Such a smaller TSL configuration, despite potential accuracy trade-offs, could yield better overall performance through reduced access latency and energy consumption, as discussed in Section D.2). Our experiments demonstrate LLBP-X’s potential to compensate for accuracy loss with smaller predictors while enabling higher overall performance.

## VIII. RELATED WORK

Jiménez [20] proposed prefetching prediction counters into a small-and-fast structure using old global histories, applied to a simple gshare predictor with a fixed history length. In contrast, LLBP-X uses program context to prefetch TAGE metadata for multiple patterns of varying history lengths in a single lookup, performing this operation at most once per context rather than every cycle.

Prior work [6], [43], [44] explored hiding predictor latency through ahead-pipelining, which starts prediction early with incomplete history and pre-computes multiple predictions in parallel. While recent work [6] addressed energy concerns of ahead-pipelining by replacing multi-ported predictors with “dual-tag” pattern matching, ahead-pipelining still requires accessing the entire predictor every cycle. The high power draw from frequency accessing a large predictor prevents scaling up predictor capacity. LLBP-X shares the same philosophy of prefetching based on incomplete history but uniquely accesses the main storage *only* when program context changes and the context is tracked in the CD. These fewer lookups are the key feature of LLBP-X that unlocks the larger storage capacity prior approaches cannot achieve.

A number of works, including [9], [32] have focused on “Impossible to predict” data-dependent branches, and proposed precomputing branch outcomes by executing the necessary instructions ahead of time. These works are orthogonal to our work, which seeks to improve accuracy for branches that can be predicted accurately given sufficient predictor capacity.

Recent work, Whisper [22], analyzed branch prediction in server workloads and identified that TAGE-SC-L’s limited capacity constrains prediction accuracy. To address this limitation, Whisper proposed a cross-layer solution combining offline application profiling, ISA extensions, and specialized prediction circuits. While effective, this approach is highly invasive as it requires extensive cross-layer support and depends on having representative application profiles. In contrast, LLBP-X takes a purely microarchitectural approach that requires neither profiling nor ISA modifications.

Several approaches have leveraged machine learning to target hard-to-predict branches [50], [53]. However, these solutions require offline profiling, architectural modifications, and operating system support, making them difficult to adopt compared to pure microarchitectural solutions.

## IX. CONCLUSION

Modern server workloads challenge branch prediction due to their expanding instruction footprints, while predictor capacities remain latency-constrained. Although the recently introduced LLBP design shows a promising way forward by augmenting TAGE with decoupled metadata storage, it exhibits significant limitations compared to conventional predictors. This work shows that pattern contention and duplication are the primary sources of inefficiencies. We address these issues through dynamic contextualization, substantially improving LLBP’s accuracy with minimal hardware modifications.

## ACKNOWLEDGMENT

The authors thank the anonymous reviewers, Andreas Sandberg from Arm, as well as 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 are grateful to Caeden Whitaker, Mike Jennrich, and Matt Sinclair from the University of Wisconsin-Madison for helping with an initial gem5 implementation of LLBP, as well as Phillip Assmann from the Technical University of Munich for his significant effort in improving the model’s correctness during his thesis work. This research was generously supported by the University of Edinburgh, Arm and by EASE Lab’s industry partners and sponsors including Huawei, Intel and Cisco.

## APPENDIX

### A. Abstract

This artifact provides the framework used in our work to analyze LLBP and develop LLBP-X, available at <https://github.com/dhschall/LLBP-X>. The repository contains two main components.

The first is a trace-based evaluation framework, including an LLBP-X implementation compatible with simulators such as ChampSim or CBP. We also provide a lightweight simulator that models only the branch predictor, enabling rapid design exploration. It includes implementations of LLBP-X, LLBP, and TAGE-SC-L.

The second component is a gem5 implementation of LLBP-X and LLBP. We reuse as much of gem5’s existing TAGE code as possible to simplify integration and reduce redundancy, though structural differences in gem5 may introduce minor model discrepancies.

The artifact is intended to support future research by documenting how to build and run both models. While it illustrates the full workflow for the trace-based framework—obtaining sources and traces, building and running predictors, and analyzing results—it does not aim to reproduce every result presented in the paper.

Finally, Section H provides instructions for integrating LLBP-X into gem5 and running a simple hello world example.

### B. Artifact check-list (meta-information)

- **Compilation:** C++ compiler with C++20 standard.
- **Model:** LLBP-X and LLBP models for CBP/ChampSim and gem5.
- **Data set:** Traces are available at <https://zenodo.org/doi/10.5281/zenodo.13133242>. Download traces using the supplied script.
- **Metrics:** Branch mispredictions (MPKI)
- **Experiments:** Use the provided script to evaluate branch MPKI reduction.
- **How much disk space required (approximately)?:** ~25GiB for the traces + ~5GiB for gem5
- **How much time is needed to prepare workflow (approximately)?:** ~1-2 hours. Mostly to download the traces.
- **How much time is needed to complete experiments (approximately)?:** Each configuration takes between 15 and 45 minutes to simulate, depending on the model, the benchmark, and the used machine. Running all 84 configurations on a 32-core machine should take about an hour. Building gem5 takes another ~15 min. Running the simple hello world example just a few seconds.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Workflow automation framework used?:** GitHub actions are used for continuous integration tests.
- **Archived (provide DOI)?:** <https://doi.org/10.5281/zenodo.17807918>

### C. Description

1) *How to access:* The source code is publicly available on GitHub (<https://github.com/dhschall/LLBP-X>) or Zenodo (<https://doi.org/10.5281/zenodo.17807918>).

2) *Hardware Dependencies:* The LLBP-X framework can be used on any system with a general-purpose CPU and at least 36 GiB free disk space to store the traces and build gem5.

3) *Software Dependencies:* The framework requires a C++ compiler with C++20 standard, CMake 3.22 or above, and depends on the boost library. It is tested on Ubuntu 20.04 and 22.04 with GCC v9.4.0 and v11.4.0 and Clang v11.0 and v14.0. For plotting the graphs, the matplotlib library and a Jupyter notebook are used.

Running gem5 experiments requires all dependencies to build gem5. Refer to the gem5 documentation for details [15]

4) *Data sets:* The server traces used to evaluate LLBP are available on Zenodo (<https://zenodo.org/doi/10.5281/zenodo.13133242>) and can be downloaded using the supplied script. Ten traces were collected while running server applications on gem5 in full system mode, while four traces were obtained from the Google Workload Traces ([https://dynamorio.org/google\\_workload\\_traces.html](https://dynamorio.org/google_workload_traces.html)). All traces are converted into the ChampSim [16] format.

### D. Installation

To obtain the source code, install all dependencies, and build the simulator, execute the following commands.

```
# Clone the LLBP-X repository:  
git clone https://github.com/dhschall/LLBP-X.git  
  
# Install dependencies  
sudo apt install -y cmake libboost-all-dev \  
build-essential pip parallel wget
```

```

pip install -r analysis/requirements.txt

# Compile the simulator
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Debug ..
cd ..
cmake --build ./build -j $(nproc)

```

The traces used to evaluate LLBP-X can be obtained from Zenodo with:

```
./scripts/download_traces.sh
```

### E. Simulation

To reproduce the MPKI reduction of LLBP-X, similar to Figure 12, use the following command to launch the simulations for all branch predictor models and benchmarks.

```
./scripts/eval_all.sh
```

### F. Evaluation and Expected Results

The results folder should contain 84 statistic files, with six files per benchmark. Use the Jupyter Notebook `./analysis/mpki.ipynb` to parse the files and evaluate the results. Press Run All to execute all notebook cells. This should produce two PDFs plotting absolute MPKI and MPKI reduction. The MPKI reduction should be similar to the reduction reported in Figure 12 in Section VII-A.

### G. Experiment Customization

The `README.md` file provided with the repository contains additional information on the code structure along with instructions to run individual experiments. The source code is well-documented, with clear and concise comments that explain variables, parameters, and methods. Most of the configuration parameters for LLBP can be found in the `LLBPXConfig` struct within the file.

### H. gem5 Experiments

This section explains how to integrate and use the LLBP-X gem5 model to support future research. 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 [15], [23], [35]—we do not provide the full system setup used in our evaluation. Instead, we focus on demonstrating how to integrate the model into gem5 and how it operates using a minimal hello-world example.

This section explains how to integrate and use the LLBP-X gem5 model to support future research. For simplicity, we do not provide the full-system setup or server workloads used in our evaluation; for these, we refer readers to the gem5 documentation and existing frameworks from prior work that present these details [15], [23], [35].

The artifact does not include the full gem5 source tree. Instead, it provides only the new LLBP-X model and the modified TAGE-SC-L components needed for integration. The steps below outline how to obtain gem5, integrate the models, build gem5, and run a simple Hello World example using the different branch predictors.

**1) Setup:** Refer to the gem5 documentation [15] 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 inside the `LLBP-X` directory, or for convenience, run the `./scripts/setup_gem5.sh` script.

```

# Clone gem5
git clone git@github.com:gem5/gem5.git
# Check out the compatible version
cd gem5
git checkout v25.1.0.0
# Copy the LLBP-X models into the gem5 source tree
cp -r ../gem5models src/cpu/pred/
# Apply patch
git apply ../scripts/llbpX.patch
# Build gem5
scons build/ARM/gem5.opt -j $(nproc)

```

**2) Run Experiment:** To test the models, a simple configuration script is provided in the `scripts` folder (`se-llbp.py`). You can run the example using LLBP-X as the branch predictor with:

```
cd gem5
./build/ARM/gem5.opt \
    ./../scripts/se-llbp.py --bp=LLBPX
```

Alternatively use:

```
./scripts/run_all_gem5.sh
```

to simulate the three models—LLBP-X, LLBP, and TSL64k—and print their branch misprediction counts. For the simple hello-world example, all three models should report around 56 mispredictions.

### I. Methodology

Submission, reviewing and badging methodology:

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

## REFERENCES

- [1] 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.
- [2] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kaney, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, “AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers,” in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*. ACM, 2019, pp. 462–473.
- [3] 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.
- [4] 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>
- [5] 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 Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

- [6] L. Cai, A. Deshmukh, and Y. Patt, "Enabling ahead prediction with practical energy constraints," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 559–571. [Online]. Available: <https://doi.org/10.1145/3695053.3730998>
- [7] B. Cohen and M. Subramony. (2024) Next generation “zen 5” core. [Online]. Available: [https://hc2024.hotchips.org/assets/program/conference/day2/24\\_HC2024.AMD.Cohen.Subramony.final.pdf](https://hc2024.hotchips.org/assets/program/conference/day2/24_HC2024.AMD.Cohen.Subramony.final.pdf)
- [8] Datadog, “The University of Utah,” 2024. [Online]. Available: <https://www.cloudlab.us/hardware.php>
- [9] A. Deshmukh, C. Cai, and Y. Patt, “Timely, efficient, and accurate branch pre-computation,” ser. MICRO '24, 2024.
- [10] 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>
- [11] Dynamorio. (2024) Google workload traces. [Online]. Available: [https://dynamorio.org/google\\_workload\\_traces.html](https://dynamorio.org/google_workload_traces.html)
- [12] M. Evers, “Improving branch prediction by understanding branch behavior,” Ph.D. dissertation, University of Michigan, USA, 2000.
- [13] O. Foundation. (2024) Introduction to node.js. [Online]. Available: <https://nodejs.org/en/learn/getting-started/introduction-to-nodejs>
- [14] gem5 developers. (2022) gem5. [Online]. Available: <https://github.com/gem5/gem5/releases/tag/v22.0.0.1>
- [15] ——. (2025) gem5 documentation. [Online]. Available: <https://www.gem5.org/documentation/>
- [16] N. Gober, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, “The Championship Simulator: Architectural Simulation for Education and Competition,” 2022.
- [17] T. P. Group. (2024) Fastcgi process manager (fpm). [Online]. Available: <https://www.php.net/manual/en/install.fpm.php>
- [18] Intel, “Intel Xeon Gold 5418N Processor,” 2024. [Online]. Available: <https://www.intel.com/content/www/us/en/products/sku/232392/intel-xeon-gold-5418n-processor-45m-cache-1-80-ghz/specifications.html>
- [19] C. Jacobi, A. Saporito, M. Recktenwald, A. Tsai, U. Mayer, M. M. Helms, A. Collura, P. kin Mak, R. J. Sonnelitter, M. A. Blake, T. Bronson, A. O’neill, and V. K. Papazova, “Design of the IBM z14 microprocessor,” *IBM J. Res. Dev.*, vol. 62, no. 2/3, pp. 8:1–8:11, 2018.
- [20] D. A. Jiménez, “Reconsidering Complex Branch Predictors,” in *Proceedings of the 9th IEEE Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2003, pp. 43–52.
- [21] R. E. Kessler, “The Alpha 21264 microprocessor,” *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.
- [22] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, “Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications,” in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 19–34.
- [23] E. Lab. (2022) vswarm-u: Microarchitecture for serverless workloads. [Online]. Available: <https://github.com/ease-lab/vSwarm-u>
- [24] C. Lam. (2024) Lion cove: Intel’s p-core roars. [Online]. Available: <https://chipsandcheese.com/p/lion-cove-intels-p-core-roars>
- [25] 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>
- [26] C.-K. Lin and S. J. Tarsa, “Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions,” in *Proceedings of the 2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 228–238.
- [27] 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. Hanindhitto, 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. Shingarow, M. D. Sinclair, T. Ta, R. Thakur, G. Travagliini, 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.
- [28] S. Mahar, H. Wang, W. Shu, and A. Dhanotia, “Workload behavior driven memory subsystem design for hyperscale,” 2023. [Online]. Available: <https://arxiv.org/abs/2303.08396>
- [29] P. Michaud, “A PPM-like, Tag-based Predictor,” *J. Instr. Level Parallelism*, vol. 7, 2005.
- [30] ——, “Best-offset hardware prefetching,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 469–480.
- [31] A. H. plc, “Arm neoverse v2 platform: Leadership performance and power efficiency for next-generation cloud computing, ml and hpc workloads,” 2023. [Online]. Available: [HotChips2023,https://he2023.hotchips.org/assets/program/conference/day1/CP1/H2023.Arm.MagnusBruce.v04.FINAL.pdf](https://he2023.hotchips.org/assets/program/conference/day1/CP1/H2023.Arm.MagnusBruce.v04.FINAL.pdf)
- [32] S. Pruitt and Y. N. Patt, “Branch runahead: An alternative to branch prediction for impossible to predict branches,” *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021. [Online]. Available: <https://api.semanticscholar.org/CorpusID:239011545>
- [33] D. Schall. (2023) Fetch directed instruction prefetching for gem5. [Online]. Available: <https://github.com/dhschall/gem5-fdp>
- [34] ——. (2025) Speculative update for tage-sc-l. [Online]. Available: <https://github.com/gem5/gem5/pull/1854>
- [35] 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.
- [36] 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>
- [37] ——, “The last-level branch predictor,” in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ‘24. IEEE, 2024.
- [38] D. Schor. (2024) Intel details golden cove: Next-generation big core for client and server socs. [Online]. Available: <https://fuse.wikichip.org/news/6111/intel-details-golden-cove-next-generation-big-core-for-client-and-server-socs/>
- [39] A. Seznec, “A 256 kbits l-tage branch predictor,” *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, vol. 9, pp. 1–6, 2007.
- [40] ——, “A 64 kbytes isl-tage branch predictor,” in *JWAC-2: Championship Branch Prediction*, 2011.
- [41] ——, “Tage-sc-l branch predictors,” in *Proceedings of the 4th Championship Branch Prediction*, 2014.
- [42] ——, “Tage-sc-l branch predictors again,” in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.
- [43] A. Seznec and A. Fraboulet, “Effective ahead pipelining of instruction block address generation,” in *30th International Symposium on Computer Architecture (ISCA 2003), 9-11 June 2003, San Diego, California, USA*, A. Gottlieb and K. Li, Eds. IEEE Computer Society, 2003, pp. 241–252. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ISCA.2003.1207004>
- [44] 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>
- [45] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides, “Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor,” in *Proceedings of the 29th International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2002, pp. 295–306.
- [46] A. Seznec and P. Michaud, “A case for (partially) TAGged GEometric history length branch prediction,” *J. Instr. Level Parallelism*, vol. 8, 2006.
- [47] X. O. source Community, “Xiangshan open-sourceprocessor user guide, applicable to kunminghuv2r2,” XhingChang Open-source Community, Tech. Rep., 2025.
- [48] R. Suite. (2024) Renaissance suite: A modern benchmark suite for the jvm. [Online]. Available: <https://renaissance.dev/>

- [49] Supermicro, "Hyper SuperServer SYS-121H-TNR," 2024. [Online]. Available: <https://www.supermicro.com/en/products/system/hyper/1u/sys-121h-tnr>
- [50] S. J. Tarsa, C.-K. Lin, G. Keskin, G. N. Chinya, and H. Wang, "Improving Branch Prediction By Modeling Global History with Convolutional Neural Networks," *CoRR*, vol. abs/1906.09889, 2019.
- [51] WikiChip. (2024) Skylake (server) - microarchitectures - intel. [Online]. Available: [https://en.wikichip.org/wiki/intel/microarchitectures/skylake\\_\(server\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server))
- [52] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.
- [53] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2020, pp. 118–130.
- [54] J. Zhao, A. Gonzalez, A. Amid, S. Karandikar, and K. Asanovic, "COBRA: A Framework for Evaluating Compositions of Hardware Branch Predictors," in *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2021, pp. 310–320.