THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

The last-level branch predictor

Citation for published version:
Schall, D, Sandberg, A & Grot, B 2024, The last-level branch predictor. in *Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture.* Proceedings of the Annual International Symposium on Microarchitecture, Institute of Electrical and Electronics Engineers, pp. 1-16, The 57th IEEE/ACM International Symposium on Microarchitecture, Austin, Texas, United States, 2/11/24.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

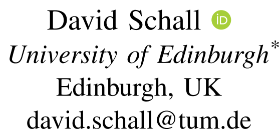
Published In:
Proceedings of the 57th IEEE/ACM International Symposium on Microarchitecture

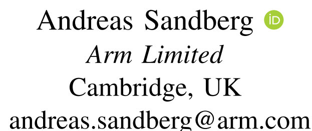
General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

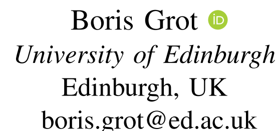
Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 13. 11. 2024

The Last-Level Branch Predictor


David Schall
University of Edinburgh[*]
Edinburgh, UK
david.schall@tum.de


Andreas Sandberg
Arm Limited
Cambridge, UK
andreas.sandberg@arm.com


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

***Abstract*—Branch prediction is crucial for modern high-performance processors, ensuring efficient execution by anticipating branch outcomes. Despite decades of research, achieving high prediction accuracy remains challenging, particularly in server workloads, where large branch working sets and hard-to-predict branches are prevalent. State-of-the-art predictors, such as the 64KB TAGE-SC-L design, experience high misprediction rates on server workloads, with 3.6-20% (9.2% on average) of execution cycles wasted due to mispredictions on a modern server CPU. While more predictor capacity can reduce mispredictions by up to 36% in the limit (with infinite storage), realizing meaningful gains in practice requires hundreds of KBs of storage, which is infeasible for a latency- and area-sensitive in-core predictor.**

This work introduces the Last-Level Branch Predictor (LLBP), a microarchitectural approach that improves branch prediction accuracy through additional high-capacity storage backing the baseline TAGE predictor. LLBP leverages the insight that branches requiring longer histories tend to span multiple *program contexts* – notionally, function calls. A given program context, which can be thought of as a call chain, localizes the branch prediction state, affording a small number of patterns per context even for hard-to-predict branches. LLBP predicts upcoming contexts and prefetches the associated branch metadata into a small in-core buffer, which is accessed in parallel with the unmodified TAGE predictor. Our results show that a 512KB LLBP backing 64KB TAGE-SC-L reduces MPKI by 0.5-25.9% (avg. 8.9%) over the baseline without LLBP.

I. Introduction

Branch prediction is a key enabling technology for today's high-performance CPUs. By anticipating branches and predicting their outcome, the branch predictor helps ensure that the front-end stays on the correct execution path and continues to fetch instructions even though a given branch may be resolved tens of cycles after it has been fetched. Improving branch prediction performance not only can improve the performance of existing processor designs but also opens the door to more aggressive microarchitectures with potentially thousands of instructions in flight [27]. Such large-window processors are only possible with highly accurate branch prediction since each misprediction incurs a pipeline flush.

Despite decades of research, achieving high prediction accuracy remains a challenge, particularly in the server domain [25]. Modern server workloads pose difficulties for effective prediction due to their large branch working set and the existence of inherently hard-to-predict branches [27]. Branches that are difficult to predict may require the predictor to track many instances of such a branch to improve accuracy. When coupled with a large branch working set, the branch predictor's storage structures become overwhelmed by the amount of branch content they need to maintain, leading to costly mispredictions.

We find that the state-of-the-art branch predictor design (64KB TAGE-SC-L) has an MPKI of 0.29-6.4 (avg. 2.91) on a suite of diverse server workloads. When these workloads are run on a recent Intel Sapphire Rapids CPU, we measure 3.6-20% (avg. 9.2%) of all execution cycles to be wasted on mispredictions, indicating that branch prediction is, indeed, a performance bottleneck.

Today's state-of-the-art branch predictors tend to be derivatives of the TAGE design, which uses an ensemble of tables, each table making predictions using geometrically-longer histories over the preceding table [29], [38]–[40]. A representative configuration in a high-end processor may use up to 30 tables and 64KiB of storage [42]. A branch is first inserted into a table using a very short history; however, if the branch is mispredicted, the branch is inserted into the next table with a longer history. Thus, a given branch may have instances present in multiple tables, and — for hard-to-predict branches — many patterns (corresponding to different histories) may need to be stored per table. We find that while the average number of histories per branch is 14, the most-mispredicted branches necessitate over 100, and up to thousands, of patterns.

One direct way to improve branch prediction performance is to simply increase predictor capacities. Indeed, we observe that a TAGE-SC-L predictor with infinite capacity available for its TAGE tables can reduce branch MPKI by 36.5%, on average, for the studied server workloads. However, for finite-capacity predictors, we find that accuracy improves very slowly as predictor size is increased from a well-provisioned baseline (e.g., 64KB). For instance, doubling the TAGE capacity from 64KB to 128KB reduces MPKI by a mere 6.4%.

Given such a trend, naively scaling up the capacity of a practical TAGE-based predictor is not a viable option given that the branch predictor is latency-, area- and power-constrained. Prior work has proposed virtualizing TAGE to enable a hierarchical predictor through a concept of paging, whereby all of the histories associated with a given region of code are colocated in a fixed-size region, and pages are moved between branch predictor levels as needed [34], [35]. Problematically, fixed-size pages cannot accommodate the large skew in the number of patterns per branch, necessarily forcing a painful compromise between coverage and storage. In order to accommodate

[*]Now at Technical University Munich

branches with a large number of patterns, the page size must be large; however, given that the vast majority of branches require a small number of patterns, large pages waste capacity. Our studies indicate that nearly a third of the opportunity offered by infinite-capacity TAGE comes from improved accuracy on just the top 0.8% most-mispredicted branches, which require numerous histories. Moreover, because of the high storage cost and the bandwidth overhead associated with a paged branch predictor, prior work concluded that a practical design can page only one TAGE table (out of 30 in a state-of-the-art design) [35].

The only work targeting branch predictor performance in the context of server workloads proposed a cross-layer solution combining costly off-line profiling and analysis, novel circuits for prediction, and ISA support [25]. While shown to be effective in reducing branch mispredictions, the proposal is highly invasive due to the required cross-layer support.

In this work, we introduce the *Last-Level Branch Predictor (LLBP)*, a purely microarchitectural approach to improve branch prediction accuracy through additional storage capacity in the form of a new predictor level. Our work tackles the key challenges in designing an effective branch predictor hierarchy, which includes the ability to accommodate a large skew in the number of patterns per branch, the bandwidth associated with transferring patterns between the levels of the branch predictor, and the timeliness of accessing the new (large-but-slow) level.

LLBP is enabled by a new insight into branch behavior based on the observation that branches that require a longer history length tend to span multiple *program contexts* (e.g., function calls). Program contexts can be represented as a sequence of unconditional branches; e.g., jumps, calls, returns. Our insight is that for branches that require many histories, a program context can be used to localize the branch so that only a small number of patterns need to be maintained per context. For instance, with a context depth of 32, 95% of all branches require just nine patterns or fewer per context.

Leveraging this insight, LLBP deploys large-capacity storage backing an unmodified TAGE-based predictor. Internally, LLBP is organized into small regions, each corresponding to a unique global context and storing a handful of patterns associated with that context. LLBP is accessed via a per-core *context directory*, which is indexed by a hash of the global context. A core-side buffer stores the branch patterns associated with the current and recently-accessed contexts. The buffer is accessed in parallel with the baseline TAGE predictor, and the longest pattern among the two predictors is used to make the final prediction. Finally, in order to avoid accessing LLBP in the latency-critical prediction path, LLBP uses storage-free context prediction to anticipate upcoming contexts and prefetch the associated patterns.

To summarize our contributions:

- We corroborate prior work by showing that server workloads have a high branch prediction MPKI of 0.15-6.3 (avg. 2.53). On a contemporary server, 3.6-20% (avg. 9.2%) of all execution cycles are wasted due to branch mispredictions.

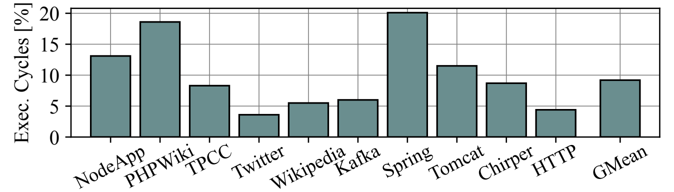


Fig. 1: Execution cycles wasted due to conditional branch mispredictions.

- We find that branch predictor accuracy can be improved by up to 36.5%, on average, through additional storage capacity. However, significant amounts of additional storage are needed to realize meaningful gains. Moreover, the additional storage must be able to accommodate the wide distribution in the number of patterns per branch.
- We demonstrate that *program context* (e.g., function call sequence) can be used to localize a small set of patterns needed to predict a given branch instance, even for branches that need a large number of patterns for high prediction accuracy.
- We introduce the *Last-Level Branch Predictor (LLBP)*, a new microarchitectural approach for improving branch prediction accuracy using program contexts. LLBP features high-capacity secondary storage backing the baseline TAGE predictor. LLBP tracks a small number of patterns per program context, which can be prefetched to the core through storage-free context prediction. A 512KB LLBP backing 64KB TAGE-SC-L reduces branch MPKI by 0.5-25.9% (avg. 8.9%) over a baseline without LLBP.

II. Motivation

A. Branch Prediction on a Modern CPU

Modern high-performance processors feature deep and wide pipelines and enable out-of-order execution across hundreds of instructions [4], [14], [37], [53]. Deep speculation supported by a highly-accurate branch predictor is vital for enabling such aggressive designs by ensuring that the processor remains on the correct path and the pipeline stays busy. A single mispredicted branch can take multiple tens of cycles to detect and, correspondingly, waste the work of hundreds of subsequently fetched instructions, ultimately degrading performance, wasting energy and increasing carbon footprint [5], [23].

To understand the importance of branch prediction on the performance of a modern CPU, we conducted experiments on a recent Intel Sapphire Rapids server using a set of representative contemporary server workloads. Our workloads and setup are detailed in Section VI. We collect CPU performance metrics from the loaded server and categorize execution cycles akin to Intel's Top-Down methodology [55]. We are interested in how many of the executed cycles are wasted because the branch predictor mispredicted the direction of a conditional branch.

Figure 1 presents the results of this study and shows 3.6-20% (9.2% on average) of the overall execution cycles are

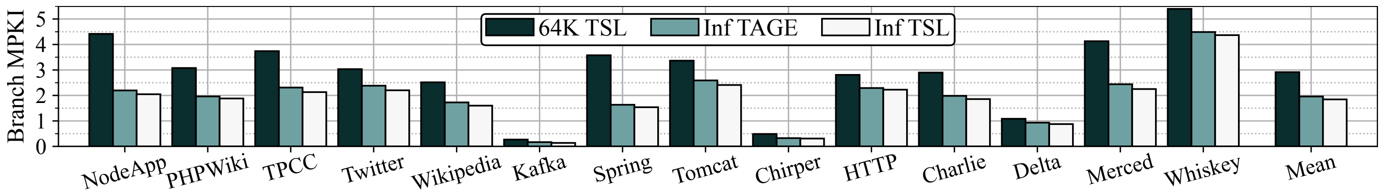


Fig. 2: Branch Mispredictions Per Kilo-Instructions (MPKI) for different configurations of TAGE-SC-L.

wasted because of conditional branch mispredictions. The results reveal that branch mispredictions present a major performance bottleneck and a significant source of inefficiency, even for the latest generation high-performance server CPU.

B. State-of-the-art in Branch Prediction

To date, TAgged GEometric history length (TAGE) [29], [38]–[40], [42], [47] is considered the most accurate branch prediction algorithm [27]. TAGE is the winner of the last four championships [30]–[32], [43] on branch prediction, and variants of TAGE are implemented in modern commercial CPU's [1], [19], [33]. It is, therefore, the focus of this work.

TAGE is based on *prediction by partial matching* (PPM) [29] and identifies correlations by comparing patterns in global history with previously observed patterns. A pattern is a specific path of control flow transitions (branches) leading to a branch. TAGE uses multiple predictor tables Ti, each containing patterns with different, geometrically increasing lengths of global history. A state-of-the-art TAGE implementation uses up to 30 prediction tables, with the longest history length of up to 3000 bits [42]. A table entry consists of a *tag*, a signed *prediction* counter, and a *useful* bit. The sign of the prediction counter determines the predicted direction and the useful bit guides replacement.

To match a pattern (i.e., to make predictions), a hash function is used to compress the branch address (PC) and the corresponding global history of a table into a table index and a tag to match. When the tag matches, the prediction counter determines the likely direction based on previous occurrences of the same pattern. In case of multiple pattern matches, the pattern with the longest history determines the prediction (referred to as *provider*), and if none of the tags match, an untagged bimodal table (BIM) is used as a fall-back. If the prediction from a matched pattern was wrong, TAGE creates a new pattern with a longer history length.

To recycle unused patterns, TAGE uses the *useful* bit to determine a pattern's usefulness. A pattern is identified as "useful" if the provider was correct and the alternative prediction, a matching pattern with a shorter history or the BIM, was incorrect. Conversely, if two patterns provided a correct prediction, the pattern using a longer history is not needed and its useful bit is cleared.

In the latest version, 64KiB TAGE-SC-L, the core TAGE predictor is augmented with two auxiliary components to improve its accuracy. The first is the *statistical corrector* for hard-to-predict and statistically biased branches, and the second is the *loop predictor* for predicting loop exits.

C. TAGE in the Limit

Because prior research has shown that more capacity generally improves prediction accuracy [20], [21], [25], [41], we ask in this section; can the accuracy of TAGE, as the most accurate and practicality-proven branch predictor algorithm to date be improved by providing it with more capacity.

We use ChampSim [15] to study the same workloads as in Section II-A and, additionally, a set of production traces provided by Google [11]. Section VI details our experimental methods.

As our baseline, we use the winner of the latest branch prediction championship, 64KiB TAGE-SC-L [42], and compare it against a version of the same predictor with unlimited capacity. To create the unlimited capacity version, we do not modify the hash functions or increase the number of tables in TAGE. Instead, we tag each pattern with the branch PC and allow unbounded associativity. For brevity, we will refer to the two versions as *64K TSL* (the baseline 64KiB TAGE-SC-L) and *Inf TSL* (infinite capacity TAGE-SC-L).

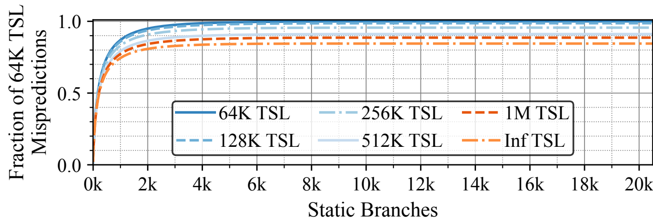
To understand whether accuracy gains come from the main TAGE or the auxiliary predictor components (the statistical corrector and the loop predictor), we consider one additional configuration where we only provide unlimited capacity to the pattern tables in TAGE while keeping the capacity of the auxiliary components unmodified. We refer to this configuration as *Inf TAGE*.

The comparison is presented in Figure 2 and shows branch mispredictions per kilo-instructions (MPKI) for the three evaluated configurations. The results show 64K TSL has a misprediction ratio of 0.29-6.4 (avg. 2.91) MPKI. The unlimited version (Inf TSL) reduces mispredictions by 36.5% to 0.14-4.37 MPKI (avg. 1.55 MPKI), revealing a significant opportunity in increasing the prediction structures.

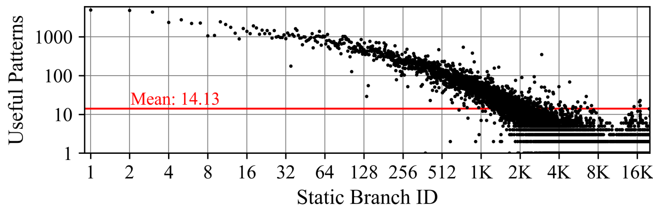
We further observe that the Inf TAGE configuration, which only increases the capacity of the TAGE tables but not the auxilliary components, captures nearly all of the opportunity of the Inf TSL configuration, reducing mispredictions by 14-54% (avg. 31.9%). This represents 87% of the opportunity offered by Inf TSL, indicating that increasing the capacity of the TAGE tables holds the biggest opportunity in reducing branch mispredictions.

D. Understanding the Branch Working Set

We next study TAGE's working set to gain insights into which branches benefit most from additional capacity to subsequently guide our branch predictor optimizations.



(a) Cumulative mispredictions.



(b) Useful Patterns per static branch for Inf TSL.

Fig. 3: Mispredictions and the number of useful patterns per static branch for *Tomcat* sorted by mispredictions. Mispredictions are normalized to 64K TSL.

We simulate TSL with progressively larger storage budgets ranging from 64K to 1M, as well as Inf TSL (as described earlier), and trace the number of mispredictions and the number of unique patterns that provide a *useful* prediction for individual static branches. Recall that a pattern is considered *useful* when it provides a correct prediction while the alternative prediction from a shorter matching pattern or the bimodal predictor is incorrect.

In Figure 3, we present the results for Tomcat, which exhibits trends representative of the other benchmarks. In both graphs of the figure, static branches are sorted by the number of mispredictions in the 64K TSL configuration.

Mispredictions: Figure 3a shows that most mispredictions come from a relatively small number of branches. Out of 20.5K unique branch PCs executed in this workload, 160 branches (0.8% of the branch working set) are responsible for 40% of all mispredictions. Our results corroborate similar findings for SPEC workloads [27], which showed that an average of 6 branches are responsible for up to 50% of all mispredictions in a typical workload of SPEC. Note, however, that SPEC workloads have considerably small branch working sets (average of 4.4K branches) than our studied server applications, indicating a similar overall trend but with a considerably smaller branch footprint than modern server workloads.

Figure 3a shows that, compared to 64K TSL, Inf TSL reduces total branch mispredictions by 35%. Roughly a third of this opportunity (31% of the overall reduction) comes from reducing the mispredictions associated with the 160 most-mispredicted branches. From this result, we draw two conclusions. First, to get any meaningful improvement over 64K TSL, it is essential not to ignore these most-mispredicted branches. And secondly, focusing *only* on these branches is equally insufficient, since two-thirds of the opportunity lies elsewhere.

The additional lines in Figure 3a show TSL configurations with storage capacities between 64K and Inf. We observe that mispredictions decrease very slowly with additional predictor capacity. Doubling the capacity from 64K to 128K reduces mispredictions by only 6.4%. Further doublings from 256K to 1M add reductions of 7.1% (256K), 7.3% (512K), and 4.1% (1M) over the previous configuration. These findings corroborate prior work [27] in demonstrating that significantly increasing the storage capacity is the primary means to improve TAGE's accuracy.

Patterns per branch: Figure 3b presents a similarly skewed distribution for the number of useful patterns per branch. TAGE requires orders of magnitude more patterns for the most mispredicted branches compared to the rest. Although the average number of patterns per branch is 14.13, the 100 most-mispredicted branches have over 100, and up to 9500, useful patterns per branch. In the following, we will use the term *complex* branches to refer to the top most-mispredicted branches, distinguishing them from the other branches.

Take-away: Increasing the capacity of TAGE can significantly reduce branch mispredictions, including for the most-mispredicted branches. However, a capacity increase by multiple factors is needed to get meaningful reductions.

The branch predictor working set is highly skewed, with the most-mispredicted branches requiring one or more orders of magnitude more useful patterns for accurate prediction as compared to the rest of the branches.

III. Jacking Up the Branch Predictor

In order to accommodate a large branch working set with an associated massive number of patterns, it may be tempting to simply scale up the TAGE predictor. A naïve approach to doing so would be to simply scale-up the TAGE predictor's capacity. Another option is to create a hierarchical TAGE design where the recently-used patterns are stored in a small-and-fast predictor and less-recently-used patterns are in a large-but-slow structure. In this section, we show that both options are unsatisfactory.

A. Up-scaling TAGE

The naïve way to provide more storage is to increase the sizes of the prediction tables in TAGE. However, this comes at the cost of increased latency and energy consumption.

Since the branch predictor is on the front-end's critical path, predictions need to have a low latency to avoid adversely impacting performance [21], [22]. Due to the high clock frequencies and the complexity of modern predictors, today's sophisticated branch predictors cannot provide predictions in a single cycle [1], [24], [46]. Instead, they use a scheme called *overriding* where a fast but low-confidence predictor provides an early prediction that is confirmed or disagreed by a more accurate, but larger and slower, predictor a few cycles later [12], [33], [57].

While overriding can help to reduce the gap between latency and accuracy, it has fundamental limits [22]. As the size

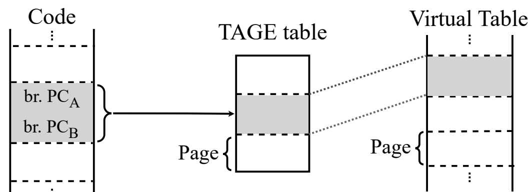


Fig. 4: Virtualized TAGE table using paging scheme [35]

of the predictor increases, its latency also increases, which reduces the benefit of more accurate prediction. There exists a point where the higher accuracy provided by the larger design fails to outweigh the associated latency penalty [21]. For example, Jimenez [21] demonstrated that there are cases where a 512KiB predictor achieves 11% lower IPC than the same predictor with only 32KiB when accounting for latencies.

In addition to an increased latency, larger predictor structures consume more energy. Using CACTI [6], we estimate that increasing the size of TAGE by 8x would increase its energy draw by over 4.5x (Section VII-D).

Take-away: Naïvely up-scaling the structures of TAGE is not attractive because the benefits of higher branch predictor accuracy are undermined by the latency and energy penalties of the larger prediction structures.

B. Hierarchical TAGE

To overcome the trade-offs between capacity, latency, and energy consumption, computer architects have long been building hierarchical designs for other structures, such as caches and BTBs [8], [17], [52]. By exploiting spatial and temporal locality in the access patterns to instructions and data, only the currently active part of the working set can be kept in a small and fast structure that is back-filled from a larger but slower structure.

However, in contrast to instruction caches or BTB entries, branch predictor metadata is not purely a function of the instruction address. Instead, TAGE uses a hash function to combine branch PC with global branch history of various lengths to index its predictor tables. This creates two key challenges to overcome for a hierarchical branch predictor design: First, the hash function scatters entries for branches that are close-by in the instruction stream, thus completely destroying spatial locality among branches. This makes it challenging to efficiently prefetch groups of branches from secondary branch storage. Secondly, combining a branch PC with branch history often leads to multiple patterns per branch. Our characterization in Section II-D shows massive variance in the number of patterns per branch, with most branch PCs having a small number of patterns but some with thousands of patterns.

Prior work [34], [35] proposed addressing the first challenge using a paging scheme, depicted in Figure 4. By modifying the hash function of the table index, patterns from branches within a region of code are collocated onto a *page*. Collocated as a page, patterns can be swapped between the TAGE prediction table and a larger (virtual) table, creating a hierarchical design.

The problem with such a paging scheme is that it does not solve the second challenge of high variance in the number of patterns per branch. In the proposed design, the maximum number of patterns that can be accommodated for a given branch is bounded by the page size. If the page size is kept small, to keep storage affordable, complex branches that require a large number of patterns to be predictable will be adversely affected by the storage limit. Conversely, since the majority of branch PCs require only a handful of patterns (Section II-D), a very large page size will necessarily incur a vast storage overhead with most pages being under-utilized. Because of the high storage cost and the high bandwidth overhead associated with paging, the proposal suggested applying paging to only one TAGE table (recall that a state-of-the-art TAGE design uses up to 30 tables). If all TAGE tables were to be paged, the storage and bandwidth overheads would be amplified by a large factor, rendering the scheme impractical.

Take-away: Two challenges impede the design of a hierarchical branch predictor: (1) The aggressive use of hashing breaks patterns locality from individual branches. (2) The highly skewed distribution of patterns across individual branch PCs renders per-PC paging schemes ill-suited. To enable a viable hierarchical solution, both challenges must be addressed.

IV. Introducing Contexts

To overcome the challenges in designing a hierarchical branch predictor, namely the highly skewed branch distribution and the need to exploit spatial locality for efficiency, this work introduces the notion of a *context-sensitive branch predictor storage*. The rest of this section explains the intuition behind the idea and presents evidence to back-up the intuition. The subsequent section presents a practical design based on these concepts.

1) Intuition: Three observations motivate a context-sensitive branch predictor storage organization:

First, the number of possible patterns for a given branch increases exponentially as a function of history length. Therefore, branches with a very large number of patterns must necessarily use longer histories.

Second, longer history lengths used by TAGE span hundreds to thousands of branches. These branches likely span multiple distinct code regions across a number of functions. Transitions between the code regions take place through jumps, calls and returns. Collectively, we refer to these as a unconditional branch (UB) instructions.

Third, a sequence of UB instructions can be used to accurately identify global program state, which is similar to a call graph. Whereas a given function may be called from multiple places in the program, a call graph pinpoints the actual execution context. Similarly, a sequence of UB instructions can be thought of as a fingerprint of the current program state. We call this a *program context*.

Our hypothesis, stemming from the observations above, is that, for a given branch having multiple patterns, not

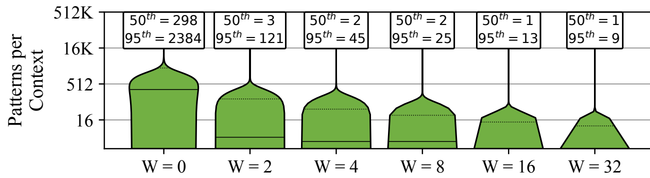


Fig. 5: Sensitivity on program context size (patterns per context) with increasing depth of previous unconditional branch instructions W.

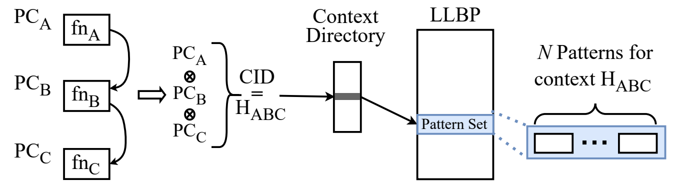


Fig. 6: High-level idea of context-sensitive branch predictor organization.

all patterns are useful in a given program context. Given a program context of sufficient depth, it can be used to localize the branch behavior such that only a small number of patterns need to be kept to make a prediction within that context. This is because the program context indirectly encodes the global (i.e, across code regions) portion of the history. Moreover, the deeper the program context (i.e., the longer the sequence of UB instructions), the more expressive it is, and therefore, the fewer patterns are needed.

2) Validation: To validate the hypothesis, we first evaluated the useful patterns of the top 10 most-complex branches for each studied workload and found that those patterns have a history length of 5-1500 branches (avg. 200). This validates the first assertion, that branches with many patterns tend to rely on long histories.

Next, we studied the ratio of conditional to unconditional branch instructions and found an average of 3.89 conditional branches between consecutive unconditional branches. Thus, for instance, a branch using a history length of 200 would span over 50 unconditional branches, likely corresponding to a number of unique code regions and functions, thereby validating the second assertion.

Lastly, we traced the number of useful patterns using the same method as in Section II-D. However, instead of counting useful patterns per unique branch, we count per unique program context formed by hashing the PCs of the previous W unconditional branches. In this study, we focus on the top 128 most-mispredicted branches, as these have the largest number of useful patterns.

The violin plot in Figure 5 presents the results as distributions of patterns per context. The left-most distribution represents the baseline without contexts ($W = 0$). Note that this is the distribution that a paging scheme [34], [35] must contend with. As the figure shows, 50% of the top 128 complex branches have more than 298 patterns per branch. At 95%, the number of patterns is 2384.

Increasing values of W effectively slice the branch history space into multiple program contexts, reducing the number of useful patterns per context by orders of magnitude. For example, using just two unconditional branches to identify a context ($W = 2$), 50% of the contexts have three useful patterns. At 95% of the distribution, the number of patterns per context is less than 40 – a reduction of nearly 60x versus a single-context organization.

Further increasing the context depth brings additional reductions in the number of patterns per context. At $W = 32$, 95% of the contexts have at most nine useful patterns per context.

The results corroborate our hypothesis that patterns exhibit locality that can be realized through the program context, expressed as a sequence of UB instructions. Doing so can be used to limit the number of useful patterns per context to just a few. We term this capability *context locality*.

3) Take-Away: Our findings on *context locality* imply that program contexts can be used to address both challenges in designing a hierarchical predictor. That is, branches with a large number of patterns can be localized such that only a small number of patterns is needed per context, and at the same time, a given context is associated with the full set of patterns needed to make predictions for it.

V. The Last-Level Branch Predictor

We present the Last-Level Branch Predictor (LLBP), a large-capacity branch predictor backing a conventional TAGE-based predictor. LLBP leverages *context locality* to enable significant predictor capacity needed to meaningfully improve the accuracy of the existing branch predictor without compromising its prediction latency. LLBP is designed around the insight that the branch predictor state exhibits inherent locality stemming from the program control flow, as explained in the previous section.

Figure 6 presents the high-level design idea. LLBP exploits context locality to break up the large branch predictor working set and reorganizes it into fine-grained *pattern sets*. Each pattern set comprises patterns associated with a specific program context (loosely corresponding to a function call chain) and is identified by a global context ID (CID). By leveraging pattern sets, LLBP uses cost-efficient hardware structures to store and access the large branch predictor working set at a fine granularity.

To overcome the longer access latencies associated with a larger structure, LLBP uses global context information to precisely identify the pattern sets required for upcoming contexts. It prefetches the upcoming patterns into a small in-core structure ahead of time, enabling low-latency predictions.

LLBP sits alongside the baseline TAGE predictor, in a way extending it. Like TAGE, LLBP stores patterns corresponding to different history lengths, and both predictors use the same partial-pattern matching algorithm to make a prediction. The final prediction is chosen based on the longest matching history length across both TAGE and LLBP. Optionally, when

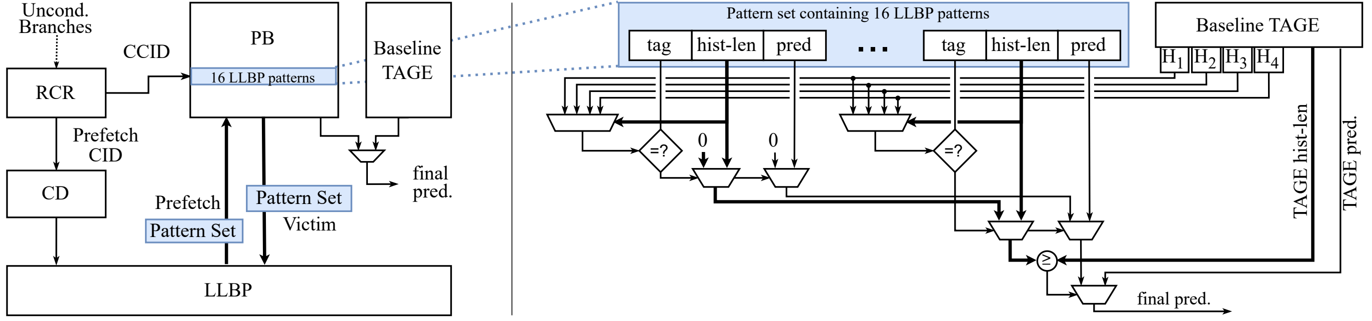


Fig. 7: LLBP architecture. Left: Design overview with the four main components along with the baseline TAGE. Right: Zoomed view on the prediction path with two pattern matches. H_x: hash of PC and GHR of increasing history length.

the accuracy of TAGE is sufficiently high, LLBP can be disabled to save power.

A. Design Overview

The left part in Figure 7 provides an overview of LLBP, which consists of four components supplementing an exiting 64K TSL branch predictor:

The *last-level branch predictor (LLBP)* is the bulk pattern set storage array. Each LLBP entry stores the complete pattern set for a given context. LLBP is optimized for storage density offers cost-effective storage space for the large working set needed to achieve highly accurate branch prediction. Due to the very effective pattern set prefetching, the LLBP can be optimized for density rather than latency. Moreover, LLBP can be virtualized into the cache hierarchy or main memory [9], thus avoiding the need for dedicated storage. We leave this optimization for future work.

The *pattern buffer (PB)* holds the pattern set corresponding to the currently active context and implements the prediction logic. It can be thought of as a mini-TAGE which maintains the prediction state and predicts outcomes for the currently active context. In addition to the pattern set of the currently active context, the PB caches the pattern sets of N most-recently accessed contexts. The pattern sets for the upcoming contexts are prefetched and also placed into the PB.

The *rolling context register* (RCR) maintains the addresses of the last unconditional branch instructions to compute the current context ID (CCID) using a rolling hash (XOR) over the context window W [54]. The CCID can be thought of as a fingerprint of the currently running region of code.

The *context directory (CD)* is used to perform an associative metadata look-up for a pattern set using its CCID. The metadata contains a valid bit, tag, a pattern set storage location, and replacement metadata. The CD is similar to the tag array in a normal data cache.

B. Making Predictions

LLBP uses the same partial pattern-matching algorithm as TAGE to make predictions [29]. However, unlike TAGE, which dynamically allocates storage to patterns, LLBP uses a single pattern set per context that holds just the set of patterns for that context. A pattern in LLBP consists of a prediction counter, a pattern tag, and a history length field. The history length field determines the length of global history that needs to be hashed to create the pattern tag[1].

The right part of Figure 7 provides a focused view of the Pattern Buffer (PB) and the prediction path. The PB's architecture is similar to TAGE, with all tag matches performed in parallel and combined using a cascade of multiplexers. LLBP differs from TAGE by storing multiple history lengths in the same pattern set (i.e., one line in the PB). The history length (hist-len) field selects the corresponding hash function (H_x) for tag matching. Each hash function is computed exactly like in TAGE by folding the Global History Register (GHR) with length $GHR[0:L_n]$ and combining it with the branch PC.

The PB is indexed by the current context ID, which is updated upon every unconditional branch. As modern CPUs typically perform one control flow redirection per cycle [1], [2], the lookup of a pattern set from the PB and pre-selection of the hash function can be performed off the critical prediction path for conditional branches.

To make a prediction, the hash functions for all allowed history lengths are computed in parallel, akin to TAGE. For each pattern in the current active contexts' pattern set, the corresponding hash is selected by the history length field and compared against the pattern tag. On a match, the sign of the prediction counter determines the predicted direction. If none of the tags match, LLBP does not provide a prediction for this cycle.

Like TAGE, LLBP may encounter multiple pattern matches during a prediction and must decide which one to use. A key element of TAGE's partial pattern-matching algorithm is the precedence given to the match with the *longest* history. This is the rationale behind TAGE's cascaded design of multiple tables indexed by progressively increasing history length hashes, as it implicitly orders the tag matches. In LLBP, however, patterns with different history lengths are likely to share a pattern set. To simplify the prediction logic, which is latency-critical, we ensure that patterns within a pattern set are stored in descending length order upon allocation. I.e., the

[1]Just like the baseline TAGE, LLBP uses a folded history register to compute the hash on the fly [29].

pattern to the left in the illustration in Figure 7 will always have a history length shorter or equal to the pattern to the right. This approach allows us to use the same multiplexer cascade as TAGE to select the longest matching pattern. Section V-D explains how the sorted order of patterns by their history length is maintained when patterns are replaced.

To arbitrate the final prediction between LLBP and the baseline predictor (TAGE), we compare the history lengths from LLBP and TAGE. Because LLBP and TAGE use the same history lengths, a 6-bit adder is sufficient to compare the table index of the TAGE match with the history length field of the LLBP match. The comparison is performed in parallel with the statistical corrector [42] and does not increase the critical path except for the final 2x1 multiplexer. If LLBP is matched on a pattern with the same or longer history length than TAGE, it overrides the baseline prediction.[2]

C. Prefetching Pattern Sets

In order to access the pattern set for the current context from the LLBP, the RCR computes a context ID, indexes the CD, and — if a match is found and not already cached in the PB — reads the pattern set from LLBP storage. The sequential accesses to CD and LLBP incur multiple cycles of latency in the critical path of reading a pattern set. To hide this latency, it is imperative to fetch the needed patterns from the LLBP ahead of time. Intuitively, that requires knowing upcoming contexts, which effectively means predicting them. However, introducing another layer of prediction for future contexts is undesirable as it would necessitate additional prediction state and logic, increasing complexity, along with a potential for mispredictions.

Instead, LLBP associates the pattern set for the current context with a *past* CID, and uses the most-recent CID for prefetching the upcoming pattern set. Figure 8 illustrates the idea. The figure shows a simplified view of the RCR, which comprises a shift register holding the PCs of the most recently executed unconditional branches. The CID of the currently active context (CCID), which indexes the PB, does not include the D most-recently executed UBs ($D = 2$ in the figure[3]). This trick effectively creates a temporal buffer of D unconditional branches between when the CIDs of upcoming contexts are known and when they become active, allowing LLBP to prefetch the upcoming pattern sets. While our sensitivity analysis in Section VII-E shows that the distance of four unconditional branches is sufficient for hiding the LLBP access latency, a different configuration and/or physical design constraints may require a different value for D.

With this organization, the only time the LLBP access latency may get exposed, and prefetches might arrive late, is in the first several cycles after a pipeline reset. Thus, if a pattern set exists for a context right after a mispredicted branch and it is not already cached in the PB, LLBP will not be able

[2]We found that overriding both TAGE + SCL prediction is almost exactly the same as just overriding TAGE and, in some cases, even better.

[3]In the evaluated design we use a hash window $W = 8$ UBs and a prefetch distance $D = 4$ UBs

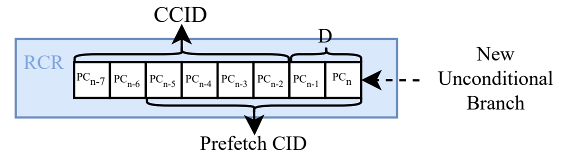


Fig. 8: Simplified view on the rolling context register (RCR) and the computation of current context ID (CCID) and prefetch context ID (Prefetch CID) with a hash window W of 6 UBs and a prefetch distance D of 2 UBs.

to provide a prediction. We study the implications of this in Section VII-A.

D. Learning Pattern Sets

While TAGE and LLBP operate almost independently for predictions, the update and training process is more coordinated. Similar to TAGE, only the providing component is updated. That is, only when LLBP overrides TAGE will the PB update the providing pattern while TAGE will cancel its update.

Upon misprediction by the providing predictor (TAGE or LLBP), LLBP allocates a new pattern with a longer history than the pattern that led to the incorrect prediction. The allocation process consists of four steps.

Step 1: This step only happens if the current context is not tracked by LLBP yet and is skipped otherwise. If no valid pattern set exists, the current context ID (CCID) is written into CD and a new pattern set is created in the PB, which potentially requires evicting an older pattern set. The new PB entry is then marked as valid.

We found that using LRU to replace pattern sets in the LLBP is a poor policy choice. Instead, the replacement policy favors keeping the pattern sets that contain many high-confidence patterns, which provide useful predictions, and evicting pattern sets dominated by low-confidence patterns. Analogous to TAGE, LLBP uses the hysteresis bits in the pattern's prediction counter to determine that pattern's confidence. The number of high-confidence patterns in a pattern set is tracked using the bits in the replacement metadata of the CD. Whenever a new pattern set needs to be inserted into the LLBP, the CD entry with the lowest number of confident patterns is chosen from the CD set indexed by the CCID.

Step 2: Given a valid PB entry, the pattern with the least-confident prediction counter value is replaced. If two patterns have the same counter value, the lower order pattern (to the left in the cascade in Figure 7) is selected.

Step 3: The victim pattern's tag and history length fields are updated with the new values, and the prediction counter is set to low-confidence taken/not-taken according to the correct direction.

Step 4: The pattern set is updated to maintain a sorted order of history lengths, effectively mimicking the order of tables in TAGE. Patterns using a shorter history are succeeded in the pattern set by patterns with a longer history, which allows

selecting the pattern with the longest history for a prediction with low logic complexity.

To balance the need for a sorted order with allocation complexity, LLBP imposes a restriction on where in the pattern set a pattern may be placed as a function of its history length. Specifically, four patterns form a bucket, each restricted to a specific range of consecutive history lengths. Lower-order buckets are assigned shorter history lengths compared to higher-order buckets. For instance, the first bucket (patterns 1-4) has history lengths 12-54, the second bucket (patterns 5-8) has history lengths 78-112, and so on.

The bucket organization offers several key advantages. It simplifies the allocation process by limiting it to a specific bucket, requiring only four patterns to be sorted by their history length. Additionally, it significantly reduces multiplexer complexity and associated delay, while decreasing the storage cost of the history length field to just two bits. Despite these simplifications, we observed only a small impact on LLBP's prediction accuracy compared to a design allowing any pattern to use *any* valid history length.

E. LLBP Details

1) Writebacks: Whenever a valid pattern set is evicted from the PB, its content is written back to the LLBP in case it was modified while in the PB.

2) Rollback of LLBP Predictions: Same with any branch prediction technique, LLBP must be able to support multiple in-fight predictions and a mechanism to roll back mispredictions [3]. To avoid polluting pattern sets with wrong information, LLBP keeps pattern sets that provide a prediction in the PB until the corresponding branch is resolved and trains patterns at commit. We found that across all benchmarks, no more than 32 LLBP predictions are ever in-flight (99.9th percentile is 12 in-flight predictions), and a PB size of 64 is sufficient to maintain all corresponding pattern sets.

While updating pattern sets happens at commit, the RCR must be updated speculatively to ensure timely prefetching. Rolling back the RCR can done in the same way as for the folded history registers in TAGE by maintaining a snapshot of the CCID and a pointer to the head of the RCR in the checkpoint of a branch [57]. Upon misprediction, the checkpointed values are used to reset the RCR.

3) Context ID Hash: To create Context IDs (CIDs), we employ a hash function that shifts each Program Counter (PC) value by twice its position in the Rolling Context Register (RCR) before XOR-ing: $CID = PC_n \oplus (PC_{n-1} << 2) \oplus \cdots \oplus (PC_{n-W+1} << (2*D))$.
By shifting each PC, we effectively encode its position into the CID, preventing repeated branch addresses from canceling each other out. This is especially beneficial for complex branches within tight loops with repeated addresses by allowing LLBP to create multiple contexts for different loop iterations.

VI. Methodology

Workloads We use a set of 14 distinct workloads obtained from various sources listed in Table I.

Application	Description
NodeApp	NodeJS online shop webserver
PHPWiki	PHP wiki web server
Kafka, Tomcat, Spring	Java DaCapo benchmark suite [7]
Finagle-chirper Finagle-HTTP	Java Renaissance suite [48]
TPCC, Twitter, Wikipedia	Java BenchBase suite [10]
Merced, Charlie, Delta, Whiskey	Google traces [11]

TABLE I: Workloads used to evaluate LLBP.

Core	4GHz, 6-way OoO, 512 ROB, 248/122 LQ/SQ
Branch Pred	64KiB TAGE-SC-L
BTB	16K entry, 8-way
Caches	32KiB 8-way L1-I, 48KiB 12-way L1-D (IP-Stride), 2MiB 16-way L2, 8MiB 16-way LLC
Prefetchers	Instructions: FDIP, Data: IP-Stride, L2: Next-line
Memory	DDR4 3200MHz

TABLE II: Parameters of the simulated processor.

Seven workloads also used by prior work on branch prediction [25] are adopted from three different Java benchmark suites [7], [10], [48]. In addition, we implemented two new workloads running standard web services. NodeApp implements a shopping website running on a NodeJS web server [13], and PHPWiki implements MediaWiki website served by a PHP-FPM content manager [16].

We collect instruction traces from those workloads using gem5 [28], which allows us to trace both user and kernel space instructions. Additionally, we utilize instruction traces provided by Google collected from four of their data center workloads [11].

Hardware Experiments: Our hardware experiments were conducted on a SuperServer SYS-121H-TNR [49] server featuring a 4th Gen. Intel Xeon 5418N Processor [18] and 128GB of DDR5 memory. SMT and frequency scaling are disabled.

Simulator Infrastructure: For simulation, we use ChampSim [15] and configured it with the parameters listed in Table II resembling a recent state-of-the-art industry baseline [4], [52]. The system is warmed-up for 100M instructions to collect statistics for 200M instructions.

We model the following branch predictor designs[4]:

64K TSL: Our baseline throughout this work is a 64KiB TAGE-SC-L [42], the winner of the latest championship on branch prediction (CBP-5) [32].

512K TSL: Is the same design as 64K TSL, but the number of table entries is scaled up by a factor of 8 (from 1K entries to 8K entries per table).

Inf TSL: 64K TSL design with an unbounded capacity. We do not increase the tags and indices widths to isolate the performance gains due to additional capacity from the algorithmic gain due to higher entropy in the hash functions. Instead, we tag each pattern with the branch PC and allow

[4]The source code of the branch predictor models and the traces are available at https://github.com/dhschall/LLBP

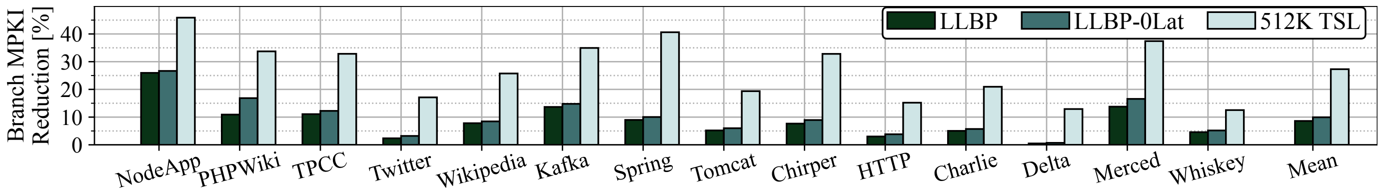


Fig. 9: Branch misprediction reduction over 64K TSL. Original MPKI shown in Figure 2.

unbounded associativity. For the statistical corrector and loop predictor, we increased all tables to 2M entries.

LLBP: Our proposal, LLBP, combined with a 64K TSL as the baseline predictor.

We use empirical studies across all workloads to determine the following parameters for LLBP: One pattern consists of a 3-bit prediction counter, a 13-bit pattern tag, and 2 bits for history length, equal to 18 bits per pattern. A pattern set comprises 16 patterns grouped in four buckets of four patterns each, for a total of 288 bits. While 64K TSL uses 21 different history lengths, LLBP uses only 16 of those split across the four buckets (Section Section V-D). We empirically found that using history lengths 12, 26, 54, 54*, 78, 78*, 112, 112*, 161, 161*, 232, 336, 482, 695, 1444, 3000 provides the best performance. The histories marked with the asterisk use the same length as the previous history but a modified hash function in analogy to 64K TSL [42].

The CD and LLBP can hold 14k pattern sets. The LLBP is directly mapped, while the CD is a 7-way set associative. A pattern set is identified by a 14-bit context ID (CCID). The 11 least-significant bits are used as a set index in the CD. The remaining 3 bits are used as CD tag. A 2-bit counter is used to replace pattern sets and stored together with the tag in the CD. The total capacity is 8.75KiB for the CD and 504KiB for LLBP. The PB caches the 64 most recent pattern sets and is 4-way set-associative with LRU replacement. Its storage cost is 2.25KiB.

The CCID is computed by hashing $W = 8$ unconditional branches using an XOR function after ignoring $D = 4$ most recently executed unconditional branches.

We model a prefetch delay of 6 cycles to cover the sequential accesses of CD and LLBP, which is based on CACTI (Section VII-D) plus one additional cycle for the logic delay. After a misprediction (BTB miss and misprediction), all in-flight prefetches get squashed before LLBP restarts prefetching.

LLBP-0Lat: LLBP configured with a zero cycle prefetch delay used to quantify the impact of late prefetches.

VII. Evaluation

A. Prediction Accuracy

We first evaluate LLBP's performance in reducing branch mispredictions. We compare LLBP against a 512K TSL, representing a TAGE-SC-L configuration with approximately the same storage budget as LLBP. Note that the latter configuration is not feasible in a practical design due to the high latency penalties associated with the massive predictor structure. Instead, we use it as a benchmark to compare against due to its proven accuracy and storage efficiency, which have been carefully tuned over the past 20 years. To understand the upper bound of LLBP without the impact of late prefetches after a pipeline reset, we also consider a configuration without access delays to LLBP's structures (LLBP-0Lat).

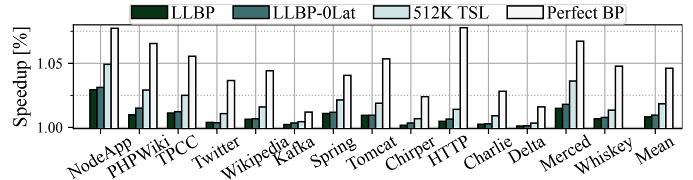


Fig. 10: LLBP's speedup over 64K TSL

Figure 9 presents branch MPKI reduction for all 14 benchmarks normalized to the 64K TSL baseline. We observe that LLBP archives a substantial reduction of branch mispredictions (MPKI) by 0.5-25.9% (avg. 8.9%). The highest MPKI reduction is achieved for NodeApp (25.9%) and Merced (13.8%).

LLBP-0Lat improves branch prediction accuracy by 0.7-26.6% (avg. 9.9%) over the baseline. Thus, the realistic LLBP configuration is within 90% of the ideal configuration with no access delay, demonstrating that LLBP's prefetching technique can effectively hide access latencies most of the time. LLBP's prefetching is least effective for PhPWiki, reaching only 66% of the ideal LLBP-0Lat. We found that PhPWiki suffers from an exceptionally high misprediction rate for indirect calls; these flush the pipeline and reset LLBP's prefetcher, thus reducing its effectiveness.

The 512K TSL reduces mispredictions by 12.5-45.9% (avg. 27.3%), outperforming LLBP by over 3x, on average. While the massive TSL design is not practical, the result does indicate a significant opportunity to improve LLBP, leaving the door open to future work to optimize the design similar to how TSL has been optimized over many generations.

B. Speedup

We evaluate LLBP's speedup over the 64K TSL baseline, achieved through increased branch prediction accuracy. We compare LLBP and LLBP-0Lat against the unrealistic but approximately equally sized 512K TSL. Additionally, we consider a perfect conditional branch predictor to establish an upper bound for branch prediction performance.

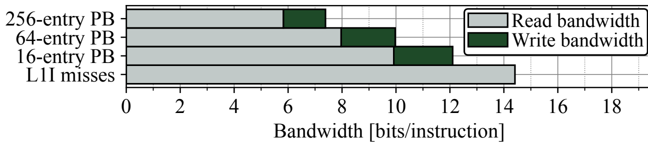


Fig. 11: LLBP's average read and write traffic across all workloads compared to the L1-I miss traffic

Figure 10 presents the results. LLBP achieves a 0.05-2.2% performance improvement over the baseline, with an average improvement of 0.63%. LLBP-0Lat's slightly higher accuracy translates into a 0.71% average speedup.

LLBP achieves 50% of the speedup observed with the 512K TSL, which improves performance by 1.26% on average. Moreover, LLBP reaches 17.5% of the performance achieved by a perfect branch predictor, showing an average speedup of 3.6%[5].

C. Transfer Bandwidth

Next, we study the communication bandwidth between the LLBP and the pattern buffer. We evaluate pattern set reads and writes, with pattern buffer (PB) sizes ranging from 16 to 256 entries. For each LLBP read/write, 288 bits are transferred.

Figure 11 shows the average results across all benchmarks, plotting bits transferred per instruction. We observe that the LLBP's read bandwidth is, on average, 9.9 bits per instruction for a PB with 16 pattern sets. Writing back modified or newly created pattern sets incurs an additional bandwidth of 2.2 bits per instruction. Thus, the writeback traffic is about 20% of the read traffic, resulting in an overall bandwidth of 12.1 bits per instruction for the LLBP. With a 64-entry PB, the combined read and write traffic bandwidth drops by 18.9% to 9.9 bits per instruction, and a 256-entry PB further brings the total bandwidth below one byte per instruction.

We chose the 64-entry PB as it represents a trade-off between hardware consumption and bandwidth reduction. Its size is 2.25KiB.

We put these numbers into perspective by comparing to the bandwidth between the L1-I cache and the L2 cache. For each L1-I miss, 512 bits are transferred. Note that the L1-I traffic includes both demand and prefetch requests. We observe that with the 64-entry PB, LLBP's read bandwidth is 8.6 bits per instruction, about 41% lower than the traffic between the L1 instruction cache and the L2 cache.

D. Latency and Energy Estimation

We estimate LLBP's latency and energy consumption using CACTI 7.0 [6] with a 22nm technology, comparing it to a 64K TSL and an unrealistic but equally sized 512K TSL. We model the CD as 7-way associative, PB as 4-way associative, and LLBP as a direct-mapped cache. We use an access width of 8b for CD accesses and 36 bytes for PB and LLBP accesses. TAGE is modeled as a direct-mapped cache, reading 42 bytes per access (21 tables * (12b tag + 3b counter + 1b useful bit)). We only model pattern tables, assuming other structures (e.g., statistical corrector) remain constant. Note that this estimation does not include energy savings due to increased prediction accuracy or the wire energy for moving pattern sets between PB and LLBP. We use a 0.25ns cycle time (4GHz clock). Results are presented in Table III.

[5]We note that a such low opportunity for perfect branch prediction is inconsistent with prior work [25] (12.4% speedup for ideal branch prediction) and our TopDown study on real hardware (Figure 1) where 9.3% of the execution cycles are lost due to misspeculation. We attribute this inconsistency to ChampSim's limited core model and leave detailed performance evaluations of LLBP for future work.

Component	Relative Access Latency	Cycles	Relative Access Energy
64KiB TSL	1	2	1
512KiB TSL	2.55	4	4.58
LLBP	2.68	4	4.44
CD	0.8	1	0.3
PB (64-enties)	0.62	1	0.25

TABLE III: Access latency and energy of LLBP structures relative to 64K TSL at 4Ghz.

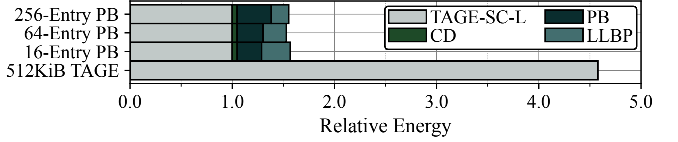


Fig. 12: Energy consumption of different designs relative to the 64K TSL

Latency: The middle column in Table III shows that increasing TAGE's pattern tables by 8x doubles the access latency from 2 to 4 cycles. LLBP lookup also requires 4 cycles. CD and PB access latencies are below 64K TSL's, fitting within a single cycle. Based on these values, we configure LLBP's simulator model with a 6-cycle prefetch delay for the sequential accesses of the CD and LLBP, plus one cycle for additional logic delay.

Energy: The last column of Table III shows access energy relative to 64K TSL. An 8x increase in 64K TSL size (to 512KB) raises energy consumption by 4.58x per access, roughly the same as fetching a pattern set from LLBP (4.45x increase). A CD and PB access consumes 31% and 25% of 64K TSL access energy, respectively.

While the energy per access to LLBP is relatively high, it is not incurred every cycle. Only the PB (0.4% of overall LLBP capacity) is accessed every cycle, as is the baseline TAGE. The CD is searched only when the context ID changes (upon executing an unconditional branch), which we found is, on average, once every 6.3 cycles. Similarly, LLBP is accessed to fetch new pattern sets only when they are not cached in the PB. We observe a LLBP access happens on average every 7.7 cycles for a PB size of 64 pattern sets.

Figure 12 plots the relative energy when access frequencies are taken into account. The figure considers LLBP with various PB sizes as well as a 512KB TSL; results are normalized to the energy consumption of 64K TSL. All LLBP structures combined consume 51-57% of 64K TSL's energy. The optimal

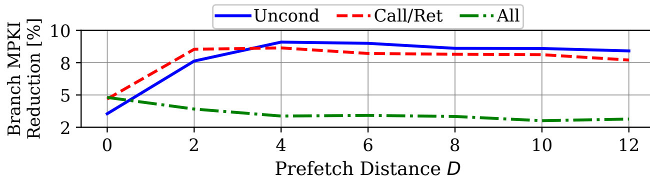


Fig. 13: CID evaluation: Sensitivity of history type and prefetch delay on the misprediction reduction. In all studies $W = 8$ branches are hashed to form the CID.

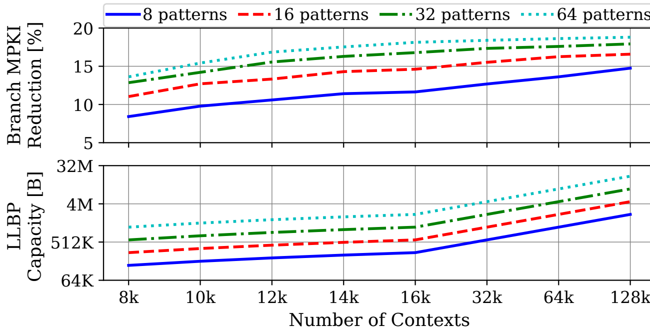


Fig. 14: Sensitivity of branch MPKI reduction and LLBP capacity on the number of pattern sets and pattern set size.

point for LLBP is with a 64-entry PB; with a larger PB, its energy consumption outweighs savings from fewer LLBP accesses. The CD consumes only 10% of all LLBP structures' energy. While LLBP with a 64-entry PB increases the energy consumption by 1.53x over 64K TSL, a 512K TSL increases the energy consumption by over 4.5x.

E. Program Context Sensitivity

LLBP uses global context information to group patterns in sets. We evaluate the effectiveness of three different branch types in forming the CID: (1) all unconditional branches (*Uncond*) – as advocated in this paper, (2) only calls and returns (*Call/Ret*), and (3) all branches (*All*).

Figure 13 shows that with no prefetch distance, all history types perform relatively poorly within 3.5-4.8% MPKI reduction. The reason is that without some prefetch distance, the LLBP is unable to provide a prediction in time.

Ignoring a few of the most recent branches to add prefetch distance affects accuracy differently. Using the Uncond or Call/Ret history increases accuracy, as both can effectively capture the global control flow with few branches and predict upcoming contexts. Uncond history (20% of all branches) performs best with $D = 4$, resulting in a maximum MPKI reduction of 8.9%. Call/Ret history (14% of all branches) is too coarse and, as a result, less effective at capturing global control flow.

For the All history, it's the opposite; increasing D increases mispredictions. The reason is that adding conditional branches adds too much noise to capture the global control flow.

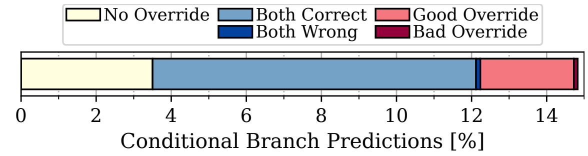


Fig. 15: Average breakdown of LLBP predictions. LLBP provides for 14.8% of the dynamic conditional branch executions a prediction.

F. Evaluating Pattern Sets

We next study the sensitivity of the LLBP to the number of pattern sets and the pattern set size (patterns per pattern set) on branch misprediction reduction and the LLBP storage consumption. For this study, we use the LLBP-0Lat model, a fully associative context index (CI), and no pattern bucketing as described in Section V-D to avoid any bias from associativity and prefetching. We also use a 31-bit context tag (instead of 14 bits) to enable more than 16K contexts. We measure the reduction in branch mispredictions compared to the 64K TSL baseline across a range of 8K to 128K pattern sets and four different pattern set sizes, 8, 16, 32, and 64. The LLBP capacity is calculated for each configuration using the empirically determined parameters presented in Section VI.

Figure 14 presents the average MPKI reduction across all 14 benchmarks and the corresponding LLBP capacity.

We observe that with 16K contexts and a pattern set size of 8 patterns, LLBP achieves an 11% MPKI reduction. Doubling the pattern set to 16 results in an additional 2.6% MPKI reduction. However, further increases in pattern set size yield diminishing returns: a pattern set size of 32 gains only 1.4% further reduction, while 64 provides less than 0.1% additional improvement.

Varying the number of contexts exhibits different behavior. With 16 patterns per context, MPKI reduction scales almost linearly from 8K contexts (10.3% MPKI reduction) to 14K contexts (13.2% MPKI reduction). Beyond this point, the MPKI reduces further, albeit at a slower rate. These results demonstrate that LLBP's context-based organization effectively distributes the highly skewed branch predictor working set across numerous contexts. Consequently, the MPKI reduction is primarily a function of the storage capacity.

We found that using 16 patterns per set and 14K pattern sets a local optimum around 512 KiB total LLBP capacity and correspondingly used to configure LLBP.

G. LLBP Effectiveness

We evaluate LLBP's effectiveness by breaking down the predictions made by LLBP into different categories. The first category (*No Override*) is when an LLBP pattern matches, but its history length is shorter than a TAGE pattern match. Thus, LLBP will not override the baseline predictor. The remaining categories are predictions where LLBP matches on a pattern with the same or longer history length and overrides the baseline predictor. We classify these as *Good Override* when the baseline predictor would have mispredicted, and *Bad*

Override when the baseline predictor would have been correct, but LLBP is wrong. The final categories are predictions where LLBP overrides the baseline, but both predict the same outcome, either *Both Correct* or *Both Wrong*.

The mean breakdown over all benchmarks is presented in Figure 15. First, we observe that LLBP provides a prediction for only 14.8% of all dynamic conditional branch executions. This is expected as most branches are easy to predict for the baseline predictor, and LLBP targets branches needing a large storage capacity. In fact, 49% of all predictions are made by the simple bimodal prediction. Once LLBP provides a prediction, it overrides the primary predictor in 77% of the cases. The LLBP overrides are mostly accurate, with only 6.8% being incorrect. One reason for LLBP's high accuracy is that LLBP patterns are only valid within a specific context, which reduces destructive aliasing effects. However, in 59% of the cases, the overrides were redundant as the baseline prediction would have been the same.

While these results show that LLBP predictions are extremely accurate, they also reveal an opportunity for future work to improve LLBP's storage efficiency.

VIII. Related Work

Hardware techniques: Jiménez D. [21] highlighted the issue of long access latencies associated with large predictor structures and proposed to use old global history to pre-select prediction counters from a larger predictor table and move them into a small-and-fast table. The work applies its technique to a simple gshare predictor, which uses only one prediction table and fixed history length. In contrast, LLBP uses program context information to prefetch TAGE metadata for multiple patterns with a single look-up and matches patterns on multiple lengths of history to make predictions. Moreover, while [21] pipelines accesses to the large table, accessing it on every single cycle, our design is accessed at most once per program context switch.

Seznec et al. [44], [45] proposed ahead-pipeline predictors and showed that starting the prediction pipeline early with an incomplete history, pre-computing multiple predictions in parallel and arbitrating the final prediction based on final histories bits can overcome the predictor latency with only a small impact on accuracy. LLBP uses a similar idea to prefetch pattern sets based on a partial history into the PB. However, with ahead-pipelining, the entire branch predictor must be accessed on every cycle. In contrast, LLBP is accessed only if a pattern set exists in the CD, dramatically reducing energy consumption as shown in our evaluation in Section VII-D.

Vougioukas et al. [51] tackled the problem of cold branch predictor state due to context switches. The work proposed a small perceptron predictor as base predictor of TAGE, which can be swapped out on a context switch to dedicated storage and subsequently restored when the process is resumed. Schall et al. [36] also address cold branch predictor state for serverless functions and propose a record-and-replay mechanism to restore BTB and the bimodal predictor. In contrast, LLBP addresses the problem of large branch working sets for long-running workloads.

Kumar et al. [26] observed that program control flow has global (notionally, across functions) and local (within functions) aspects, and used this observation to optimize BTB design. LLBP also leverages the dichotomy between global and local control flow, but in a different way and for a different purpose – namely, designing a branch prediction hierarchy.

Software techniques: Recent work called Whisper has focused on the branch prediction problem in the context of server workloads, observing that capacity constraints of TAGE-SC-L stand in the way of higher prediction accuracy [25]. In response, the work proposed a cross-layer approach for enhancing the accuracy of branch prediction that combines offline application profiling and analysis, ISA extensions, and specialized circuits for prediction. The approach is highly invasive given the required cross-layer support, and is also reliant on having a representative profile to be effective. Our work attacks the same problem as Whisper and corroborates their findings regarding branch predictor capacity constraints under modern server workloads. However, LLBP is a purely microarhcitectural approach that requires no profiling and no ISA support.

Other techniques have focused specifically on hard-to-predict branches using ML-based methods [50], [56]. These techniques also necessitate offline profiling, architectural extensions and OS support, rendering them more challenging to adopt as compared to pure microarchitectural approaches. These techniques also do not directly help with the problem of large branch working sets prevalent in server workloads.

IX. Conclusion

In modern high-performance processors, branch prediction is essential for efficient execution, but it faces accuracy challenges particularly on server workloads, which feature large branch working sets and complex branch behavior. On these workloads, the state-of-the-art TAGE-SC-L predictor suffers from high misprediction rates, wasting significant execution cycles as a result. This work introduced the Last-Level Branch Predictor (LLBP), which addresses the capacity challenge by adding high-capacity storage to the TAGE predictor. A key insight underpinning our design is to use the notion of program context, such as a function call chain, to localize the branch predictor state to a small number of patterns for a given context. To hide the access latency to LLBP, our design uses a storage-free program context predictor to anticipate upcoming contexts and prefetch the associated branch metadata from LLBP.

Acknowledgment

The authors thank the anonymous reviewers as well as the members of the EASE Lab at the University of Edinburgh for their valuable feedback on this 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 TAGE-SC-L and design LLBP, available at https://github.com/dhschall/LLBP. The repository contains:

1) Source code for both TAGE-SC-L and LLBP
2) Instructions for compiling the code, running experiments, and analyzing results
3) Server traces used to evaluate LLBP

By only modeling the branch predictor — without the complexity of the entire CPU — the framework is a fast and easy way evaluate different branch predictor configurations and explore the design space of LLBP. Thus, the provided artifact is intended to empower future research and does not aim to reproduce the exact results presented in our work.

B. Artifact check-list (meta-information)

- **Compilation:** C++ compiler with C++20 standard.
- **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)?:** $\sim$ 25GB
- **How much time is needed to prepare workflow (approximately)?:** $\sim$ 0.5 hours. Mostly to download the traces..
- **How much time is needed to complete experiments (approximately)?:** Each configuration takes between 15-45min to simulate depending on the model, the benchmark, and the used machine. Running all 56 configurations on a 32-core machine should take less than an hour.
- **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://zenodo.org/doi/10.5281/zenodo.13197409

C. Description

1) How to access: The source code is publicly available on GitHub (https://github.com/dhschall/LLBP) or Zenodo (https://doi.org/10.5281/zenodo.13197409).

2) Hardware dependencies: The LLBP framework can be used on any system with a general-purpose CPU and at least 32 GiB free disk space to store the traces.

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 matplotlib library and a Jupyter notebook is used.

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). All traces are converted into the ChampSim [15] format.

D. Installation

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

```
# Clone the LLBP framework from GitHub repository:
$ git clone https://github.com/dhschall/LLBP.git

# Install dependencies
$ sudo apt install -y cmake libboost-all-dev build-
    essential pip parallel
$ pip install -r analysis/requirements.txt

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

E. Experiment workflow

1) Traces: The traces used to evaluate LLBP can be obtained from Zenodo with:

```
$ ./utils/download_traces.sh
```

F. Simulation

To reproduce the MPKI reduction of LLBP, similar to Figure 9, use the following command to launch the simulations for all branch predictor models and benchmarks.

```
$ ./utils/eval_all.sh
```

G. Evaluation and expected results

The results folder should contain 56 statistic files, with four 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 9 in Section VII-A. Note that for this paper, we integrated LLBP with ChampSim to obtain more precise timing and performance results. Therefore, the results of the simulator are expected to differ slightly from those presented in the paper.

H. 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 comments explaining variables, parameters, and methods. Most of the configuration parameters for LLBP can be found in the `LLBPConfig` struct within the `llbp.h` file.

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 *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, 2020, pp. 27–39.

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

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*. IEEE Computer Society, 2003, p. 423. [Online]. Available: https://doi.org/10.1109/MICRO.2003.1253246

[4] Andrei Frumusanu, "Golden Cove Microarchitecture (P-Core) Examined," 2024. [Online]. Available: https://www.anandtech.com/show/16881/a-deep-dive-into-intels-alder-lake-microarchitectures/3

[5] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers." in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 462–473.

[6] 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.

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

[8] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. R. Prasky, and A. Saporito, "Two level bulk preload branch prediction." in *Proceedings of the 19th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2013, pp. 71–82.

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

[10] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "Oltpbench: 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

[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] A. Frumusanu. (2024) Apple's humongous cpu microarchitecture. [Online]. Available: https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2

[15] 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.

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

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach, 5th Edition.* Morgan Kaufmann, 2012.

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

[21] D. A. Jiménez, "Reconsidering Complex Branch Predictors." in *Proceedings of the 9th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2003, pp. 43–52.

[22] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors." in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2000, pp. 67–76.

[23] S. Kanev, J. P. Darago, K. M. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. M. Brooks, "Profiling a warehouse-scale computer." in *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, 2015, pp. 158–169.

[24] R. E. Kessler, "The Alpha 21264 microprocessor." *IEEE Micro*, vol. 19, no. 2, pp. 24–36, 1999.

[25] 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)*, 2022, pp. 19–34.

[26] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the Front-End Bottleneck with Shotgun." in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIII)*, 2018, pp. 30–42.

[27] 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)*, 2019, pp. 228–238.

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

[29] P. Michaud, "A PPM-like, Tag-based Predictor." *J. Instr. Level Parallelism*, vol. 7, 2005.

[30] T. J. of Instruction-Level Parallelism. (2011) 2nd jilp workshop on computer architecture competitions (jwac-2): Championship branch prediction. [Online]. Available: https://jilp.org/jwac-2/

[31] T. J. of Instruction-Level Parallelism. (2014) 4th jilp workshop on computer architecture competitions (jwac-2): Championship branch prediction (cbp-4). [Online]. Available: https://jilp.org/cbp2014/

[32] T. J. of Instruction-Level Parallelism. (2016) 5th jilp workshop on computer architecture competitions (jwac-2): Championship branch prediction (cbp-5). [Online]. Available: https://jilp.org/cbp2016/

[33] 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://hc2023.hotchips.org/assets/program/conference/day1/CPU1/HC2023.Arm.MagnusBruce.v04.FINAL.pdf

[34] M. Sadooghi-Alvandi, "Exploring virtualization techniques for branch outcome prediction," Master's thesis, University of Toronto, 2011.

[35] M. Sadooghi-Alvandi, K. Aasaraai, and A. Moshovos, "Toward virtualizing branch direction prediction." in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 455–460.

[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] 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/

[38] 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.

[39] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *JWAC-2: Championship Branch Prediction*, 2011.

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

[41] A. Seznec, "Exploring branch predictability limits with the mtage+sc predictor," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[42] A. Seznec, "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. (2024) Branch prediction research. [Online]. Available: https://team.inria.fr/pacap/members/andre-seznec/branch-prediction-research/

[44] 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

[45] 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

[46] 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)*, 2002, pp. 295–306.

[47] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction." *J. Instr. Level Parallelism*, vol. 8, 2006.

[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] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: mitigating branch predictor side-channels," in *25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019*. IEEE, 2019, pp. 466–477. [Online]. Available: https://doi.org/10.1109/HPCA.2019.00058

[52] WikiChip. (2024) Golden cove - microarchitectures - intel. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/golden_cove

[53] WikiChip. (2024) Sunny cove - microarchitectures - intel. [Online]. Available: https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove

[54] Wikipedia. (2024) Rolling hash. [Online]. Available: https://en.wikipedia.org/wiki/Rolling_hash

[55] 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.

[56] 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)*, 2020, pp. 118–130.

[57] 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.