

The Path to BTB Storage Efficiency Goes Through Tags

Roman K. Brunner

Norwegian University of Science
and Technology (NTNU), Trondheim, Norway

Rakesh Kumar

Norwegian University of Science
and Technology (NTNU), Trondheim, Norway

Abstract—Modern server workloads exhibit multi-megabyte code footprints that overwhelm front-end structures such as the branch target buffer (BTB), leading to frequent misses and significant performance degradation. BTB misses are particularly costly, as they trigger pipeline flushes, expose fetch latency, and reduce the effectiveness of fetch-directed instruction prefetching. To mitigate these effects, modern processors deploy increasingly large BTBs, often requiring hundreds of kilobytes of storage. However, this trend is becoming untenable due to area, energy, and sustainability constraints.

Recent work has aggressively optimized branch target representation, shifting the storage bottleneck from targets to tags. We observe that tags now dominate BTB storage and, more importantly, exhibit substantial redundancy as only 23.7% of entries contain unique tags in a 2K-entry BTB, decreasing to just 1.1% at 32K entries. This paper leverages this insight to improve BTB storage efficiency. We propose LiteBTB, a storage-efficient BTB design that eliminates tag redundancy by storing each unique tag once in a centralized structure called the Unique Tag Buffer (UTB) and replacing per-entry tags with compact pointers. To avoid increasing access latency, LiteBTB accesses the UTB and BTB in parallel. Our evaluation shows that LiteBTB reduces BTB storage by 5.4% to 13.1% compared to the state-of-the-art BTB-X, for storing 2K to 32K branches while providing similar performance.

Index Terms—branch target buffer, microarchitecture, storage efficiency

I. INTRODUCTION

The multi-megabyte code footprints of modern server applications overwhelm the storage capacity of core front-end structures such as branch target buffer (BTB), instruction caches (L1-I), etc., leading to frequent misses. BTB misses are particularly detrimental to performance as they result in not only throwing away tens of cycles worth of work done on the wrong path, but they also expose pipeline fill latency. Further, frequent BTB misses also reduce the effectiveness of fetch directed instruction prefetching (FDIP) [1], which is commonly employed in commercial CPUs [2]–[4]. Indeed, recent studies demonstrate that BTB misses hurt performance more severely than misses in L1-I, and that an ideal BTB delivers substantially greater performance gains than an ideal L1-I [5]–[7].

To mitigate the performance cost of BTB misses, commercial processors feature massive BTBs with 100s of thousands of entries requiring 100s of KB of storage. In addition, BTB capacities are growing at an alarming rate across successive

V 1b	Tag 12-14b	Type 2b	Repl-Policy 3b	Target 46b
---------	---------------	------------	-------------------	---------------

Fig. 1: Typical composition of a conventional BTB entry.

CPU generations. This trend is evident across diverse computing domains, spanning high-performance server-class CPUs to energy-constrained mobile processors. For example, IBM’s server-class z16 employs a 260K-entry BTB [8] which is over 10× larger than the 24K-entry BTB in zEC12 [9] just four generations earlier. Similarly, Samsung Exynos M6, a mobile CPU, dedicates 529KB of storage to its BTBs, an increase of nearly 6× over the 90KB budget of Exynos M1/M2 [10].

Although these enormous BTBs are very effective in minimizing misses, they have several downsides. First, they incur large storage and area overheads. Second, the slowing down of Moore’s law makes the trend of increasingly larger BTBs untenable. Finally, their increasing area footprint results in elevated embodied carbon emissions, rendering them environmentally unattractive [11]. Therefore, it is critical to investigate techniques that reduce BTB storage overhead without hurting their hit rates.

The storage requirements of a BTB are conventionally dominated by the branch targets, as is evident from the composition of a typical BTB entry shown in Figure 1. Consequently, recent work [12]–[14] has focused on optimizing target representation to control the growing storage and area costs of BTBs. The state-of-the-art design, called BTB-X [12], replaces the branch targets with branch offsets as it observes that the offsets are typically much smaller than full targets and require much less storage. Further, since not all offsets are the same size, BTB-X features differently sized BTB entries to accommodate the offset size variability.

We observe that the branch target representation in the recently proposed BTB designs is so optimized that it is the tags that now account for the bulk of BTB storage requirements rather than the targets. Specifically, branch targets constitute only 36% of the storage budget in BTB-X, whereas the share of tags has risen to 43%. In contrast, in conventional BTB, the targets account for 72% of the storage compared to the tags contributing only 19%. We further observe that at the current BTB scales, saving even a single tag bit per BTB entry would result in substantial storage savings. For example, saving one bit per entry in the 260K entry IBM z16 BTB would save

more than 32KB of storage, which is on par with typical L1 cache capacities.

This work seeks to reduce tag storage overhead to maximize BTB storage efficiency. To that end, our key insight is that the number of unique tags present in the BTB is significantly smaller than the number of BTB entries, i.e. there is a large degree of tag redundancy in BTB entries. Moreover, the fraction of entries containing unique tags decreases sharply as BTB capacity increases. Our analysis reveals that in a 2K-entry BTB, only about 23.7% of entries correspond to unique tags, whereas this fraction falls to nearly 1.1% in a 32K-entry BTB. These results imply that BTBs exhibit a high degree of tag redundancy, which can be exploited to reduce their storage requirements.

Building on this insight, we propose to de-duplicate tags by storing each unique tag only once in a dedicated tag-store. Each BTB entry then stores a *pointer* to the desired tag in the tag-store, rather than the tag itself. Since the number of unique tags is small, the tag-store requires significantly fewer entries than the BTB, allowing tag pointers to be encoded with fewer bits than the full tags and thereby yielding substantial storage savings. A naive realization of this approach, however, can increase BTB access latency. In particular, such a design first accesses the BTB to obtain the tag pointer and subsequently performs a second access to retrieve the tag from the tag-store. This serialized lookup - pointer fetch followed by tag read - extends the critical path relative to a conventional BTB that stores tags directly within each entry.

This work introduces LiteBTB, a storage-efficient BTB design that stores each unique tag only once in a centralized tag-store, called Unique Tag Buffer (UTB), and replaces tags in BTB entries with pointers to UTB. To eliminate the serialization between pointer retrieval and tag read, as in the naive design, LiteBTB orchestrates the UTB such that it can be accessed in parallel with the BTB. Since many BTB entries can point to a single UTB entry, UTB evictions can lead to stale pointers in BTB that may result in aliasing. We propose a lazy scrubbing policy that minimizes aliasing without the exorbitant cost of proactive scrubbing. This work makes the following key contributions:

- We observe that tags now dominate the BTB storage requirements, not the branch targets.
- We make a key insight that there is a large degree of tag redundancy in BTBs and the redundancy increases with the BTB size. In particular, about 23.7% of entries contain unique tags in a 2K-entry BTB while only a meager 1.1% of entries are occupied by unique tags in a 32K-entry BTB.
- We introduce LiteBTB that tames the tag redundancy by storing a unique tag only once in a dedicated unique tag buffer (UTB) and keeps the access latency in check by accessing UTB and BTB in parallel.
- We propose an efficient and lazy BTB scrubbing scheme to minimize the potential aliasing introduced by UTB evictions that avoids the prohibitive cost of proactive scrubbing.
- Our evaluation shows that LiteBTB requires 5.4% to 13.1%

less storage than the state-of-the-art BTB design, i.e. BTB-X, for storing 2K to 32K branches while providing similar performance.

II. BACKGROUND

The branch prediction unit (BPU) is a critical component in modern core front-ends. It is responsible for generating a stream of PCs (program counters) that is used by the fetch unit to fetch instructions and feed the rest of the core. The application control flow, and hence PC generation, usually progresses sequentially one instruction after another up until a branch instruction is encountered, which may take the control flow to a non-contiguous instruction. Therefore, BPU has to identify control flow discontinuities, i.e. branches, before the instructions themselves have even been fetched. To do so, BPU employs branch target buffer (BTB), branch direction predictor, and return address stack (RAS).

A. Branch Target Buffer (BTB)

BTB serves three critical functions. First, it determines whether a program counter (PC) corresponds to a branch instruction before the instruction itself is fetched. Second, it identifies the type of branch instruction. Finally, it supplies the target address for the predicted taken branches. To perform these functions, as depicted in Figure 1, each BTB entry is composed of the following fields: *valid*, *tag*, *type*, *target*, *rep_policy* fields. The *valid* bit indicates whether the entry contains valid information. The *type* identifies the type of the branch instruction, i.e. conditional, call, return, or unconditional jump. The branch type determines whether or not to predict the branch direction and where to retrieve the branch target from. Specifically, a direction predictor is used to predict the direction, i.e. *taken/not taken*, of conditional branches, whereas call, return, or unconditional jumps are always *taken*. The *target* field supplies the target address if a branch is predicted to be taken. The only exception is return instructions, which obtain their targets from the return address stack. *rep_policy* decides which branch to replace when a new branch is inserted in the BTB.

The role of tag: Due to the finite BTB capacity, many branches can map to a single BTB entry (or set). Therefore, a *tag* field is used to determine whether the indexed BTB entry contains information corresponding to the PC used for indexing. To do so, when inserting a branch in the BTB, the lower-order PC bits are used to index the BTB, and the higher order bits are stored in the *tag* field of the indexed entry. On a subsequent access, the higher-order PC bits are compared to the *tag* field of the BTB entry indexed by the lower-order PC bits. A match indicates a BTB hit, which means that the indexed BTB entry contains information about the branch instruction at the current PC.

B. BTB storage budgets

To mitigate the performance penalty of frequent BTB misses arising from growing branch footprints of contemporary work-

CPU	L1 BTB	L2 BTB	Total
<i>Samsung Exynos (storage)</i>			
M1/M2	32.5KB	58.4KB	90.9KB
M3	49.0KB	110.8KB	159.8KB
M4	50.5KB	221.5KB	272.0KB
M5	53.3KB	225.5KB	278.8KB
M6	78.5KB	451.0KB	529.5KB
<i>IBM zSeries (entries)</i>			
zEC12	4K	24K	28K
z13	6K	96K	102K
z14	8K	128K	136K
z15	16K	128K	144K
z16	12K	260K	272K

TABLE I: Storage requirements and capacities of commercial BTBs [8]–[10].

loads, modern CPUs feature massive BTBs. Many contemporary CPUs feature BTBs with 10s to 100s of thousands of entries that require hundreds of KB to multiple MB of storage per core. Further, the BTB sizes are growing rapidly across successive CPU generations. As an example, Table I lists the number of BTB entries in IBM server class z-series CPUs and the BTB storage budget of Samsung Exynos mobile CPUs over several generations. As is evident from the table, the number of entries and the associated storage costs are increasing at an astonishing rate. Concretely, the number of entries in z-series CPUs increased more than ten times over merely four generations, while the storage budget in Exynos CPUs nearly doubled with each new generation, growing sixfold over a period of approximately eight years.

III. MOTIVATION

Given the rapidly increasing storage and area overheads of commercial BTBs (Section II-B), the research community is actively investigating storage-efficient designs. Since branch targets dominate the storage requirement in conventional BTBs, recent research efforts have been confined mainly to improving target representations in BTBs [12]–[15]. For example, BTB-X [12] stores target offsets, which usually require only a handful of bits to encode, instead of full target addresses, thus significantly reducing storage requirements. Further, since the offset size varies based on how far a branch instruction jumps, BTB-X features differently sized BTB entries to accommodate offset size variability.

We make a key observation that recent BTB designs have been so successful in optimizing branch target representation that a new bottleneck is now emerging from BTB storage efficiency perspective: tags are increasingly dominating BTB storage budgets. Table II reports the number of bits in each BTB entry along with the bits required for the targets and tags in a conventional BTB, PDede [13], and BTB-X. As is evident, the contribution of branch targets has dropped from 72% in the conventional BTB to only 36% in BTB-X. In contrast, the tags constitute 43% of the storage in BTB-X compared to only 19% in a conventional BTB.

	Target	Tag	Total	% Target	% Tag
Conv. BTB	46b	12b	64b	71.88%	18.75%
PDede	24b	12b	42b	57.14%	28.57%
BTB-X	10b	12b	28b	35.71%	42.86%

TABLE II: Tag, target, and total bits per entry in different BTB organizations.

A. Sizing BTB tags

The tag size offers a trade-off between BTB storage requirements and aliasing. A full tag, i.e. all program counter (PC) bits that are not used for BTB indexing, precisely determines whether the indexed entry corresponds to the accessing PC; however, it incurs high storage overhead. In contrast, a hashed tag reduces storage requirements; however, at the expense of aliasing. Specifically, multiple full tags may map to the same hashed tag, thus leading to aliasing, i.e. false hits. Such aliasing can redirect execution along an incorrect control-flow path, ultimately causing a pipeline flush and associated performance loss.

While prior work, both PDede [13] and BTB-X [12], use a tag size of 12 bits, we did a study to determine the minimum tag size(s) that approaches the performance of a full tag at different BTB capacities. For this study, we use truncation as the hash function, consistent with its use in Intel Raptor Cove and Gracemont [16]. Truncation generates the hashed tag by discarding higher-order bits of the full tag and retaining a specified number of lower-order bits.

The results, depicted in Figure 2, show that a 16-bit tag matches the performance of a full tag across all evaluated BTB capacities. However, further reducing the tag size leads to progressively increasing performance loss. Based on these results, we configure the baseline BTB-X to use the smallest tag size that achieves at least 99% of full-tag performance. Accordingly, the selected tag sizes are 13, 14, 13, 12, and 13 bits for 2K-, 4K-, 8K-, 16K-, and 32K-entry BTBs, respectively.

B. Analyzing BTB tag duplication

We make a key observation that all branches within a *region* share the exact same tag. We define a *region* as the contiguous $2^{(n+m)}$ -byte code block corresponding to a given BTB tag, where n denotes the number of bits needed to index the BTB and m is the number of least significant PC bits that are statically zero (e.g., two bits in the ARM ISA). Because basic blocks are typically small, such regions can contain a substantial number of branch instructions. For example, consider an 8-way, 8K-entry BTB: it requires 10 index bits, and with two fixed zero PC bits of ARM ISA, the region size expands to $2^{12} = 4\text{KB}$. With an average basic block size of five instructions, this region accommodates approximately 205 branches, all of which share the same tag. Since many of these branches may reside in the BTB simultaneously, the result is substantial tag duplication.

To understand the severity of this tag duplication, we study the number of unique tags present in differently sized BTBs. It

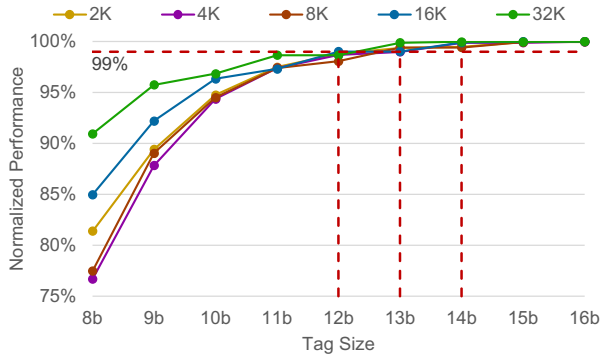


Fig. 2: Performance at different tag sizes for different BTB capacities, normalized to full tag performance.

is important to note that while a given tag cannot appear more than once within a single set, the same tag may be present across all sets of the BTB. The results of the study, plotted in Figure 3, show that only a small fraction of BTB entries are occupied by unique tags. Perhaps even more importantly, this fraction diminishes rapidly with increasing BTB capacity. Concretely, 23.7% of entries in a 2K-entry BTB correspond to unique tags, whereas in a 32K-entry BTB this number falls to nearly 1.1%. This pronounced tag duplication provides an opportunity to reduce tag overhead and increase BTB storage efficiency.

We draw two **key insights** from this analysis:

Tags dominate the BTB storage budget: Table II shows that recent advances in storage-efficient branch target representations have shifted the source of BTB storage overhead, making tags the primary contributor to overall storage overhead.

High degree of tag redundancy: Figure 3 indicates a large degree of tag redundancy that increases with BTB capacity. Specifically, the fraction of entries with unique tags drops from 23.7% in a 2K-entry BTB to 1.1% in a 32K-entry BTB.

IV. LITEBTB

Building on the insights from the previous section, we present LiteBTB, a storage-efficient BTB organization. LiteBTB is squarely focused on minimizing tag storage overhead as it increasingly becomes the dominant contributor to the BTB storage budget. Based on the insight that BTBs exhibit a large degree of tag redundancy, we propose to de-duplicate the tags by storing each unique tag only once. To realize this idea, the unique tags can be stored in a dedicated tag-store, while individual BTB entries retain only a pointer to the desired tag in the tag-store. Because the number of unique tags is small relative to the number of BTB entries, the pointer width can be substantially smaller than the tag itself, thereby reducing overall storage requirements.

While this approach lowers storage overhead, it requires additional steps during BTB lookup to retrieve the tags, since they are no longer stored directly within the BTB entries.

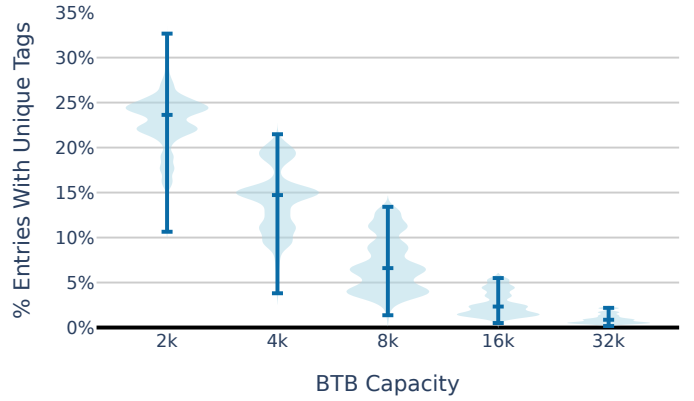


Fig. 3: Fraction of BTB entries containing unique tags.

Consequently, it may increase the BTB access latency depending on how the tag-store is organized. We therefore begin by describing a naive implementation of this idea and then introduce the proposed LiteBTB design.

A. A naive design

In a straightforward implementation of this idea, a BTB lookup would first access the BTB to obtain the tag pointer. The pointer would then be used to perform an access to the tag-store to retrieve the corresponding tag. Consequently, tag retrieval becomes a two-step process consisting of pointer lookup in the BTB followed by tag access from the tag-store.

These serialized accesses increase tag access latency compared to a conventional BTB organization, where tags are stored directly within the BTB entry and can be read in a single access. As a result, BTB hit detection is delayed, which can lengthen the critical path of the BTB lookup and potentially increase the overall BTB access latency. Such an increase in lookup latency can negatively impact front-end performance, making this design unattractive in practice despite its storage benefits.

B. LiteBTB design

The key idea behind LiteBTB is to access the tag-store - referred to as the unique tag buffer (UTB) - in parallel with the BTB, as illustrated in Figure 4. This parallelism is enabled by rethinking the BTB hit detection mechanism.

A conventional BTB detects a hit by comparing the tag stored in a BTB entry with the tag of the access PC. The naive design described in Section IV-A follows the same approach: since BTB entries store tag pointers instead of tags, it retrieves the corresponding tag from the tag-store and compares it against the tag of the access PC to determine a hit. In contrast, LiteBTB adopts a different mechanism for hit detection. Rather than comparing *tags*, it compares *tag locations*. Specifically, LiteBTB locates the tag corresponding to the access PC within the UTB and checks whether this location matches the pointer stored in the BTB entry; a match indicates a BTB hit.

LiteBTB's approach offers the advantage that UTB access does not need to wait for BTB access to complete, as the UTB

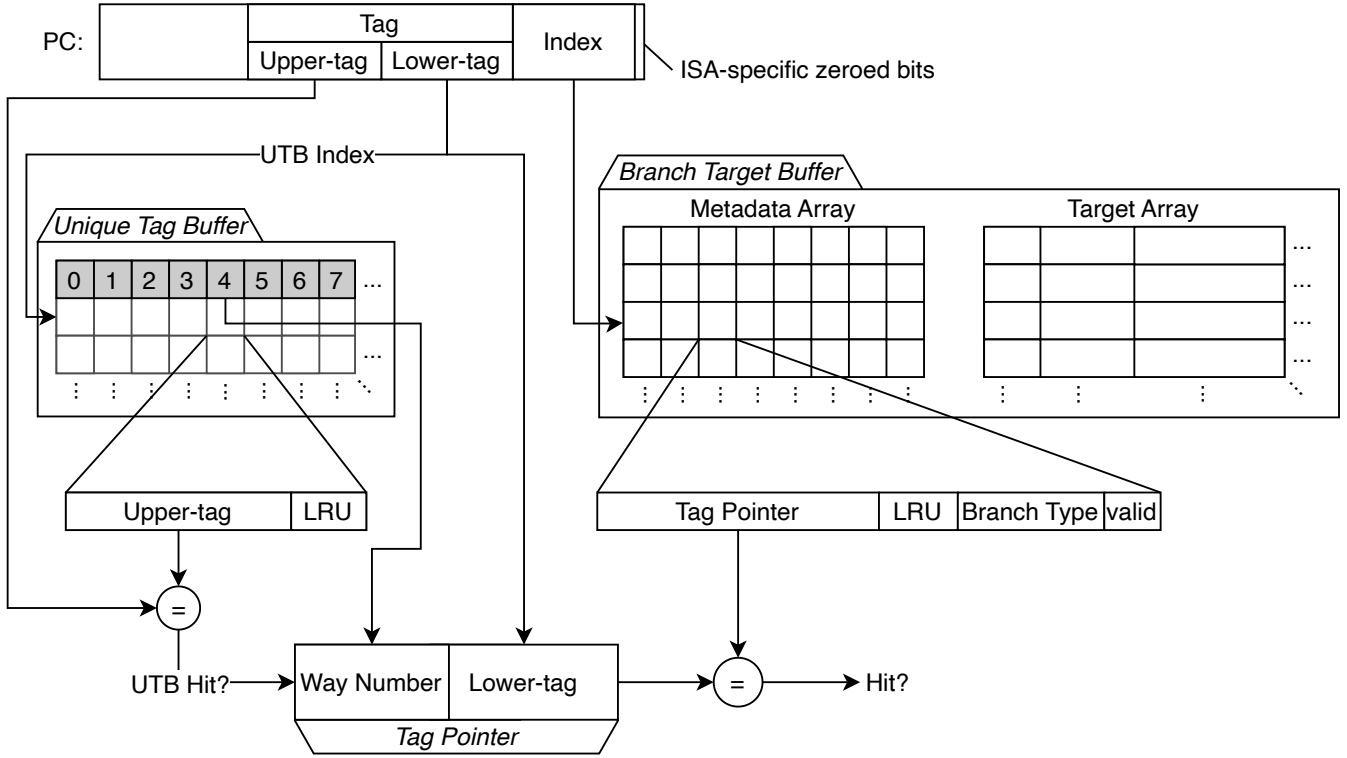


Fig. 4: LiteBTB microarchitecture.

is accessed using the *tag* corresponding to the access PC rather than tag pointers stored in BTB. Consequently, the UTB and BTB can be accessed in parallel, since the tag and index bits required for accessing these structures can be extracted from the PC concurrently.

UTB organization: For maximum flexibility in tag placement, UTB could be implemented as a fully associative structure. However, the complexity of fully associative designs scales poorly beyond a few tens of entries due to the large number of parallel tag comparisons required during lookup, which increases both latency and energy consumption. To mitigate this overhead, we instead organize UTB as a set-associative structure. Another disadvantage of fully associative UTB is that it would require storing the entire tag. In contrast, in a set associative UTB design, the tag bits are split into lower-tag and upper-tag bits as shown in Figure 4. The lower-tag bits are used for indexing, while only the upper-tag bits are stored, significantly reducing storage requirements.

LiteBTB lookup: As illustrated in Figure 4, the BTB and the UTB are accessed using the index and tag bits derived from the PC, respectively. Since the BTB is typically organized as a set-associative structure, tag pointers are read from all the ways of the indexed set. Concurrently, the lower-tag bits are used to index the UTB, and the upper-tag bits stored in all the ways of the indexed set are retrieved and compared against the upper-tag bits of the access PC. If no match is found, the tag is deemed absent in the UTB and the access results in a miss. Otherwise, the lower-tag bits are concatenated with the

way identifier of the matching entry to form a tag pointer. This pointer is then compared against the pointers read from the indexed BTB set, with a match indicating a BTB hit..

LiteBTB insertions: When inserting a new branch into LiteBTB, the UTB is first probed to determine whether the corresponding tag is already present, using the same mechanism as used for LiteBTB lookups. If the tag is not found, an entry from the indexed UTB set is evicted to make room for the new tag. The tag pointer is then inserted into the BTB together with the branch target. On UTB evictions, the BTB entries that point to the evicted UTB entry are not invalidated, as maintaining such a mapping from UTB to BTB entries would incur substantial storage and design complexity overheads.

C. Implications of UTB Misses

Because a single UTB entry may be referenced by multiple BTB entries, UTB replacements introduce secondary effects beyond conventional miss behavior.

LiteBTB misses: LiteBTB exploits the observation that multiple entries in a conventional BTB frequently share a common tag. Consequently, multiple BTB entries are likely to reference the same UTB entry in LiteBTB. As a result, evicting a UTB entry implicitly invalidates the tags of all BTB entries that reference it. These BTB entries therefore incur misses on their subsequent access, despite the branch information still being intact in BTB. We quantify the fraction of LiteBTB misses attributable to UTB evictions in Section V-E and find this fraction to be negligible.

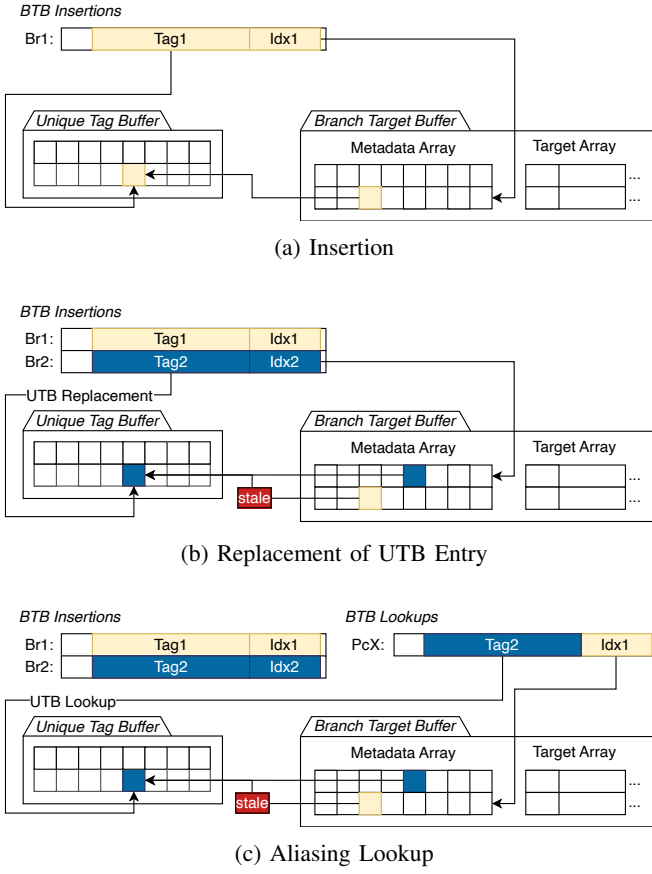


Fig. 5: UTB replacement induced aliasing

LiteBTB aliasing: As mentioned in Section IV-B, UTB evictions do not invalidate the corresponding BTB entries, leaving behind stale tag pointers that can induce aliasing. Figure 5 illustrates this scenario. A branch instruction *Br1* allocates a BTB entry at index *Idx1* that references a UTB entry storing *Tag1* (Figure 5a). Subsequently, another branch *Br2* is resolved; and then starts fetching on the correct path. If the tag corresponding to *Br2*, i.e. *Tag2*, replaces *Tag1* in the UTB, the pointer at *Idx1* becomes stale (Figure 5b). Consequently, a later lookup with a PC mapping to *Idx1* and *Tag2* results in a false hit, leading to incorrect control-flow speculation and performance degradation (Figure 5c).

A straightforward approach to eliminate aliasing is to proactively invalidate all BTB entries that reference a UTB entry upon its eviction. However, such proactive BTB scrubbing requires maintaining explicit UTB-to-BTB reverse mappings, significantly increasing design complexity. Instead, LiteBTB adopts reactive scrubbing, invalidating BTB entries only upon detecting aliasing. The aliasing is detected at the decode stage. Specifically, if the decoded instruction does not match the predicted branch type, or is not a branch, the corresponding BTB entry is invalidated. This approach avoids reverse mappings while keeping aliasing low.

The choice of proactive and reactive BTB scrubbing also impacts BTB replacements. As proactive scrubbing eagerly invalidates all BTB entries referencing an evicted UTB entry,

it ensures that invalid entries are used for replacement before any valid entry is displaced. Reactive scrubbing, in contrast, might replace useful entries as it invalidates BTB entries lazily when aliasing is detected. However, this lazy invalidation also has an advantage: if a tag is reinserted into the UTB at the same position (i.e. UTB way) as before, e.g. *Tag1* replaces *Tag2* in the example above, then the tag pointers in all BTB entries that previously pointed to this tag become instantly valid again. Therefore, they will see a hit on their next access. Under proactive scrubbing, however, those entries would have already been invalidated and thus would miss. We evaluate the performance impact of these factors in Section V-F.

V. EVALUATION

A. Methodology

To evaluate the performance and storage savings of LiteBTB, we use ChampSim [17], [18], a trace-driven simulator that models detailed core front-end structures, including the BTB and branch direction predictors. The front-end is further augmented with a Fetch-Directed Instruction Prefetcher (FDIP) [1]. The microarchitectural parameters of the modeled processor are summarized in Table III. Each experiment consists of a 50-million-instruction warm-up phase followed by 50 million instructions for measurement.

To quantify the performance impact of LiteBTB, we use workload traces provided by Qualcomm for the first Instruction Prefetching Championship (IPC-1) [19]. These traces comprise a mix of server, client, and SPEC CPU2017 workloads. We additionally use the corrected version of the Qualcomm traces released by Feliu et al. [20]. Further, we extend ChampSim to model wrong path in the core frontend. ChampSim reads a trace file to determine the instruction stream to be fetched from L1-I. On branch mispredictions, it stops reading the trace and fetching instructions until the mispredicted branch is resolved. To model the wrong path, we modified ChampSim to use the branch prediction units (BTB, branch predictor, etc.) to generate wrong path addresses, since the trace file does not have them, and fetch them from L1-I. However, as the trace file does not have instructions corresponding to these wrong path addresses, we do not propagate them any further in the pipeline. Nonetheless, this approach does capture the impact of wrong path on the core front-end precisely as is also corroborated by recent work [21].

TABLE III: Microarchitectural parameters

Parameter	Value
Core	6-wide OoO, FDIP with 128-entry FTQ, 205 reservation stations, 512-entry ROB, 192-entry load queue, 114-entry store queue
Branch Predictor	TAGE-SC-L
L1-I	32 KB, 8-way, 4 cycles latency, 8 MSHRs
L1-D	48 KB, 12-way, 5 cycles latency, 16 MSHRs
L2	1.25MB, 10-way, 16-17 cycles latency, 32 MSHRs
LLC	30MB, 15-way, 36-37 cycles latency, 64 MSHRs

TABLE IV: Storage requirements of BTB-X and LiteBTB, and storage savings of LiteBTB at different capacities.

BTB Entries		BTB-X		LiteBTB				Storage Savings
Total (Ways x Sets)	Tag Size	Set Size	Total Size	Tag Pointer Size	Set Size	UTB Size	Total	
2K (8x256)	13b	29B	7.25KB	9b	25B	0.609KB	6.859KB	5.39%
4K (8x512)	14b	30B	15.00KB	10b	26B	1.016KB	14.016KB	6.56%
8K (8x1024)	13b	29B	29.00KB	10b	26B	0.938KB	26.938KB	7.11%
16K (8x2048)	12b	28B	56.00KB	10b	26B	1.031KB	53.031KB	5.30%
32K (8x4096)	13b	29B	116.00KB	9b	25B	0.813KB	100.813KB	13.09%

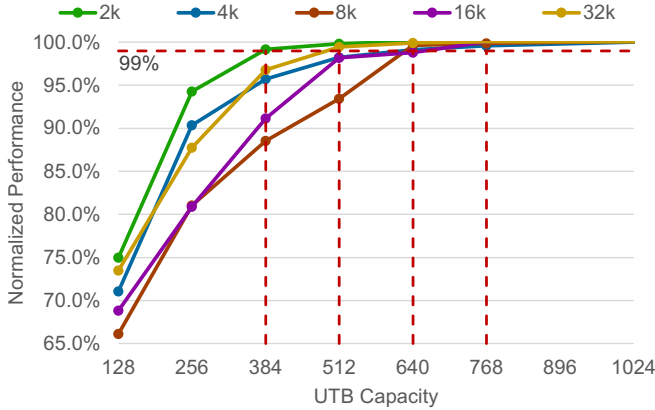


Fig. 6: LiteBTB performance across different BTB and UTB capacities, normalized to the baseline BTB-X.

B. Storage requirements

Table IV reports the storage requirements of the baseline BTB-X and LiteBTB across different BTB capacities. Both designs are 8-way set-associative, and BTB capacity is scaled by increasing the number of sets.

Common Storage: The primary distinction between BTB-X and LiteBTB lies in tag representation; all other fields are identical. As in BTB-X, LiteBTB employs asymmetric branch target encoding, where each of the eight ways uses a different target field size of 0, 4, 5, 7, 9, 11, 19, and 25 bits respectively. Due to this heterogeneity, entry sizes vary across ways; therefore, Table IV reports *set size* rather than per-entry size. The *set size* includes storage for branch targets, valid bits (1 bit), branch type (2 bits), replacement state (3 bits), and the tag-related storage aggregated over all the ways in a set. To facilitate direct comparison, Table IV also reports the tag size in BTB-X and the tag-pointer size in LiteBTB alongside the corresponding total set sizes.

Tag Storage: As discussed in Section III, BTB-X tag size requirements vary with BTB capacity (Table IV). In contrast, LiteBTB replaces tags with tag pointers, whose size depends on the UTB capacity. To determine an appropriate UTB capacity, we fix the BTB capacity and sweep UTB capacity, selecting the smallest UTB that achieves at least 99% of the baseline performance. Figure 6 shows that the optimal UTB capacity varies with BTB size: a 2K-entry BTB requires 384 UTB entries; 4K and 8K configurations require 640 entries;

TABLE V: UTB storage requirements

BTB Entries	UTB Entries	Upper-Tag	LRU	Total Size
2K (8x256)	384 (12x32)	8	4	0.609KB
4K (8x512)	640 (10x64)	8	4	1.016KB
8K (8x1024)	640 (10x64)	7	4	0.938KB
16K (8x2048)	768 (12x64)	6	4	1.031KB
32K (8x4096)	512 (16x32)	8	4	0.813KB

a 16K BTB requires 768 entries; and a 32K BTB requires 512-entry UTB to reach 99% of the baseline performance.

This non-monotonic trend, where the required UTB capacity first increases and then decreases with BTB capacity, is driven by two competing effects. First, larger BTBs experience fewer evictions and thus retain more unique tags. Second, as described in Section III, region size increases with BTB capacity, causing more PCs to map to the same tag and thereby reducing unique tags. The first effect dominates up to 16K entries, after which the second becomes dominant. Consequently, the required UTB capacity increases initially with BTB size, then decreases beyond 16K entries.

LiteBTB Storage Savings: Table IV highlights that LiteBTB provides increasingly higher storage savings over BTB-X as BTB capacity scales. For 2K-, 4K-, and 8K-entry BTBs, LiteBTB reduces storage by 5.39%, 6.56%, and 7.11%, respectively, with savings exceeding 13% at 32K entries. This trend is particularly significant given the increasing BTB capacities in modern processors.

The storage efficiency of LiteBTB stems from the reduced size of tag pointers relative to BTB-X tags. Tag pointers are four bits smaller for 2K-, 4K-, and 32K-entry BTBs, three bits smaller for the 8K configuration, and two bits smaller for the 16K configuration. Additionally, UTB overhead is negligible, particularly at larger capacities. For example, the UTB accounts for less than 1% of total storage in a 32K-entry LiteBTB. This is because the UTB is significantly smaller than the BTB, and each entry contains only a few bits (Table V).

C. Performance

Figure 7 compares the performance of LiteBTB and BTB-X under two configurations: (1) equal number of entries and (2) equal storage budget, across a range of BTB capacities. Under the equal-entry configuration, LiteBTB achieves the storage reductions reported in Table IV with negligible performance impact. Under the equal-storage configuration, it instead provides higher performance than BTB-X.

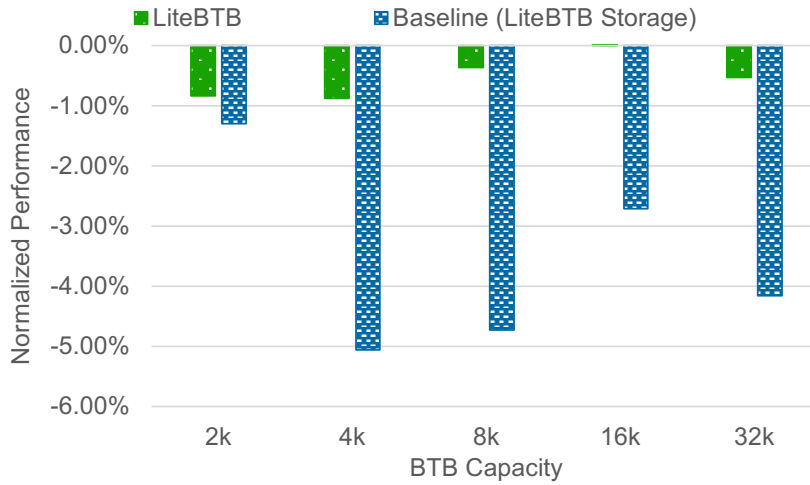


Fig. 7: Performance of different BTB designs over BTB-X.

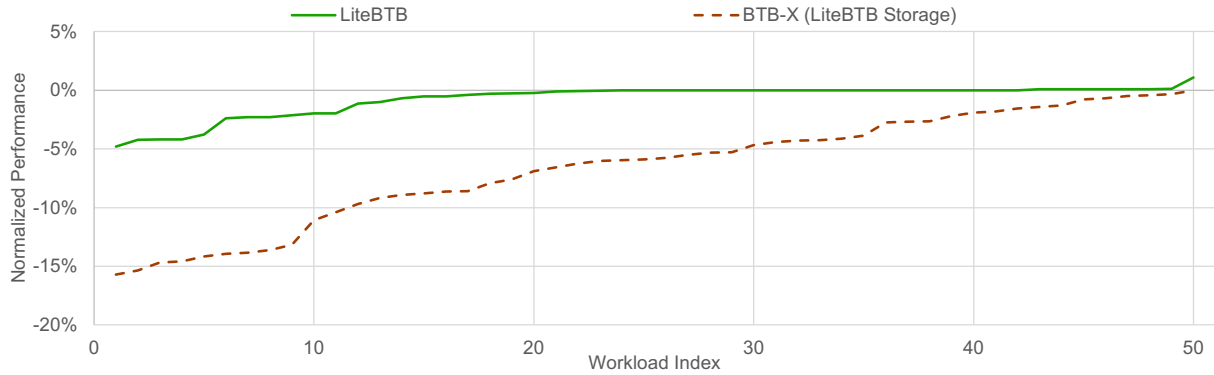


Fig. 8: Per workload performance of LiteBTB and BTB-X (LiteBTB Storage) normalized to 4K-entry BTB-X.

For the equal-storage comparison, BTB-X is constrained to the same storage budget as LiteBTB (Table IV). To enforce this constraint, we reduce the tag size in BTB-X, consistent with the design dimension targeted in this work. We denote this configuration as *BTB-X (LiteBTB Storage)*. All results are normalized to the baseline BTB-X.

Across all evaluated BTB capacities, LiteBTB incurs less than 1% performance loss relative to BTB-X, demonstrating that its storage reductions come with a negligible performance impact. In contrast, *BTB-X (LiteBTB Storage)* exhibits consistent and significant performance degradation, exceeding 5% at the 4K-entry BTB and remaining pronounced across other capacities. These results indicate that simply reducing tag size to reduce storage requirements is insufficient and leads to non-trivial performance loss. In comparison, LiteBTB achieves meaningful storage savings while largely preserving baseline performance.

Per workload performance comparison: Figure 8 presents the per-workload performance of LiteBTB and *BTB-X (LiteBTB Storage)* relative to the BTB-X baseline at the 4K-entry BTB configuration. The workloads are sorted based on speedup over baseline BTB-X for each configuration individ-

ually. Overall, LiteBTB closely tracks baseline performance, with the vast majority of workloads clustered near or above 0%, confirming that its performance loss remains negligible. In contrast, *BTB-X (LiteBTB Storage)* exhibits a significant performance loss on a number of workloads. Concretely, it experiences a performance loss of more than 10% for about 12 workloads. Furthermore, the worst-case performance loss is nearly 16% compared to only about 5% for LiteBTB. These per-workload results reinforce the aggregate trends: LiteBTB maintains robust performance across diverse workloads, whereas storage reduction via tag truncation in BTB-X leads to significant performance loss.

D. Latency, area, and energy analysis

Although the UTB and BTB are accessed in parallel, LiteBTB could incur additional latency if the UTB does not provide the tag location before the tag pointers are read out from BTB for comparison. To evaluate this, we model a 4K-entry LiteBTB using CACTI 7.0 at 22nm. CACTI reports a BTB tag pointer read latency of 0.1237ns, while the UTB returns the tag location in 0.0792ns, indicating that the UTB does not lie on the critical path. As shown in Table VI, the overall tag-hit latency, including tag comparators, of LiteBTB

TABLE VI: Timing, area, and energy requirements of BTB-X and LiteBTB

		BTB-X	LiteBTB		Savings
			UTB	BTB	
Tag Timing [ns]		0.2168	0.0792	0.2006	7.47%
Area [mm ²]		0.146	0.005	0.130	7.53%
Energy per Access [pJ]	Read	7.42	0.51	6.66	3.37%
	Write	6.99	0.58	6.11	4.29%

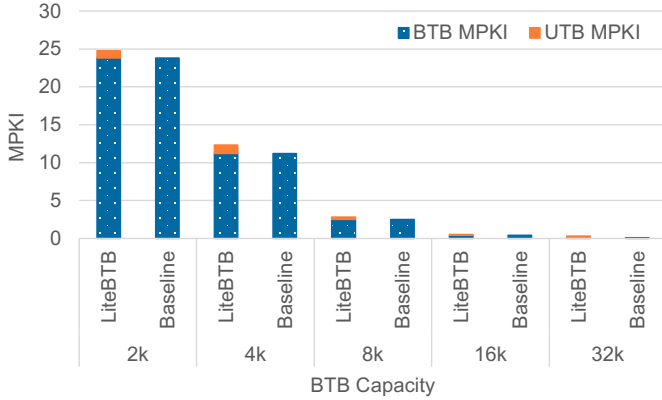


Fig. 9: LiteBTB and baseline BTB-X MPKI at different BTB capacities

is 0.2006ns, compared to 0.2168ns for BTB-X. This reduction is primarily due to the smaller comparison width of tag pointers (10b) relative to the tags used in BTB-X (14b).

Further, Table VI also shows the area and energy requirements of LiteBTB and the baseline BTB-X. The area trends mirror the storage reduction. CACTI reports that a 4K-entry baseline BTB-X occupies 0.146mm². For LiteBTB, the UTB accounts for 0.005mm² and the BTB for 0.130mm², yielding a total area of 0.135mm², thereby achieving a 7.53% area reduction relative to the baseline.

For energy, BTB-X consumes 7.42pJ per read and 6.99pJ per write access. In LiteBTB, the UTB consumes 0.51pJ per read access and 0.58pJ per write access, while the BTB consumes 6.66pJ per read and 6.11pJ per write. This results in a total energy of 7.17pJ per read and 6.69pJ per write for LiteBTB. Overall, LiteBTB reduces per-access energy by 3.37% for reads and 4.29% for writes compared to BTB-X.

E. Impact of UTB Evictions on LiteBTB MPKI

Because a single UTB entry may be referenced by multiple BTB entries, evicting a UTB entry can invalidate the effective tags of all those BTB entries, inducing misses on subsequent accesses. Figure 9 quantifies the fraction of mispredictions (i.e., LiteBTB misses resulting in pipeline flushes) caused by UTB eviction across different BTB capacities. The results show that only about 5% of LiteBTB’s MPKI originates from UTB misses.

At the same time, LiteBTB MPKI closely tracks that of the baseline BTB-X across all configurations. Overall, these results validate that the UTB contributes only a marginal

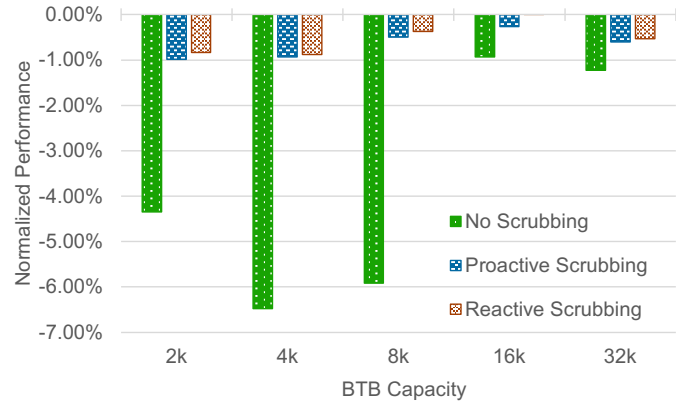


Fig. 10: Impact of different scrubbing approaches on LiteBTB performance.

fraction of total misses, while the LiteBTB maintains comparable accuracy to the baseline, corroborating the negligible performance impact observed in prior sections.

F. Evaluating BTB scrubbing policies

Figure 10 quantifies the performance impact of the BTB scrubbing policies described in Section IV-C, i.e., no scrubbing, proactive scrubbing, and the proposed reactive scrubbing, across different BTB capacities. With no scrubbing, stale tag pointers introduced by UTB evictions lead to significant performance degradation due to aliasing, with losses exceeding 6%. Proactive scrubbing mitigates aliasing by eagerly invalidating BTB entries that reference evicted UTB entries, achieving performance close to the baseline. However, it requires maintaining explicit UTB-to-BTB reverse mappings which introduces significant design complexity.

In contrast, reactive scrubbing achieves slightly better performance than proactive scrubbing without increasing design complexity, despite experiencing slightly higher aliasing (recall that reactive scrubbing invalidates a BTB entry with stale tag pointer only after it has aliased once). This improvement arises because, as discussed in Section IV-C, stale pointers in BTB entries can become useful again if their corresponding tag is reinserted into the UTB. Our analysis shows that, depending on BTB capacity, 6–7% of BTB entries invalidated by proactive scrubbing become useful again under reactive scrubbing.

G. Evaluating hash functions

As discussed in Section III-A, our baseline design adopts truncation as the hash function to generate hashed tags from full tags, following the approach used in Intel Raptor Cove and Gracemont [16]. LiteBTB further reduces tag storage requirements beyond truncated tags. Prior work has also proposed folded-XOR as a tag hashing function, as employed in Intel Skylake [16]. Folded-XOR constructs hashed tags by partitioning the full tag into equal-sized segments and XOR-ing them to produce a shorter tag.

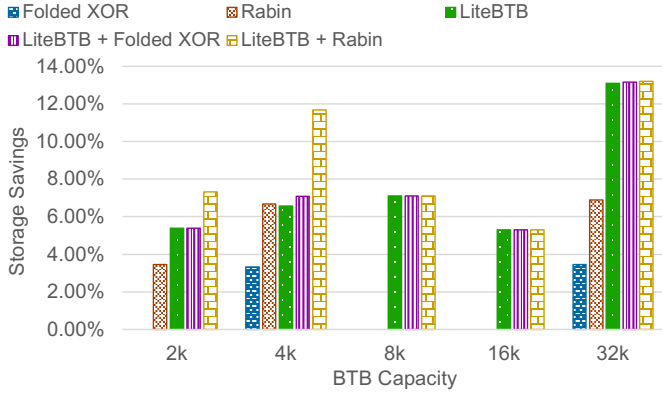


Fig. 11: Storage savings of standalone hash functions and their combinations with LiteBTB, normalized to truncation, across different BTB capacities. The LiteBTB bar represents LiteBTB with truncation.

We also examined additional hash functions from the literature. Some schemes, such as CityHash [22], produce fixed-width 32- or 64-bit outputs in publicly available implementations. Therefore, these are ill-suited for BTB tag hashing, where the required tag width is typically much smaller. Among the hash functions that can be configured to produce shorter outputs, such as Rabin [23], SHA-3 [24], and Zobrist [25], Rabin is the only that requires fewer bits (and in some cases same) than truncation, to achieve 99% of the full tag performance.

Figure 11 compares the storage savings of standalone hash functions and their combinations with LiteBTB, normalized to truncation, across different BTB capacities. The LiteBTB bar corresponds to LiteBTB built on truncation. All evaluated configurations achieve performance within 1% of the full-tag baseline. As shown in the figure, Rabin provides greater storage savings than Folded-XOR when used alone and, for a 4K-entry BTB, achieves savings comparable to LiteBTB. However, at 8K- and 16K-entry BTB capacities, neither Folded-XOR nor Rabin provides any storage savings relative to truncation

The figure also shows that combining LiteBTB with any hash function yields greater storage savings than using the hash function alone. For example, in the 4K-entry configuration, Rabin by itself reduces storage by about 6.5%, whereas combining Rabin with LiteBTB increases the savings to approximately 12%. More generally, LiteBTB provides additional savings over a hash function when the hashed tag width exceeds the number of bits required to encode the unique hashed tags resident in the BTB. In the 4K-entry BTB, for instance, Rabin produces 12-bit hashed tags, while the BTB contains only about 512 unique hashed tags. Since these can be encoded using just 9 bits, LiteBTB reduces storage by storing the unique hashed tags in the UTB and replacing each 12-bit BTB tag with a 9-bit pointer to the corresponding UTB entry.

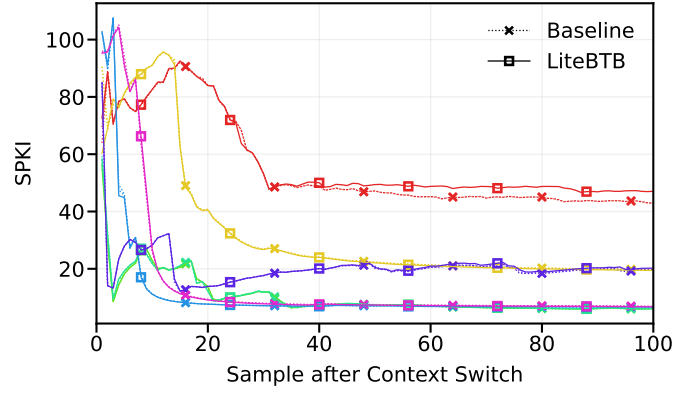


Fig. 12: Pipeline squashes per kilo instruction (SPKI) for baseline BTB-X and LiteBTB over time after a context switch.

H. LiteBTB at context switches

This section assesses how LiteBTB behaves during context switches when the unique tag working set changes. To study this effect, we execute two traces back-to-back in ChampSim and reset all statistics when the second trace begins. To capture temporal behavior, we sample execution every 32K instructions and record the number of pipeline squashes per kilo-instructions (SPKI) caused by BTB misses and aliases.

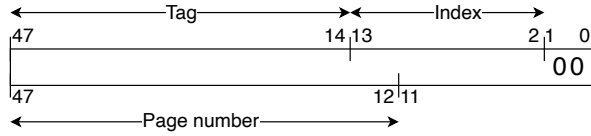
Figure 12 reports the results, with SPKI on the y-axis and execution samples over time on the x-axis. The x-axis label “0” denotes the context-switch point, i.e., when the second application begins execution. Dotted lines correspond to the baseline BTB-X, while solid lines correspond to LiteBTB. For clarity, the figure shows results for only six applications, yielding a total of 12 curves.

As shown in the figure, SPKI rises sharply immediately after the context switch due to the cold-start effects, since the BTB initially lacks entries for branches in the newly scheduled application. Also, across all workloads, LiteBTB closely tracks the SPKI of the baseline BTB-X. This is because, following a context switch, most LiteBTB misses arise from the absence of the required branches in the BTB rather than from missing tags in the UTB; the same branches also miss in the baseline BTB-X. Consequently, both designs exhibit similar SPKI. These results indicate that UTB capacity does not become a bottleneck during context switches.

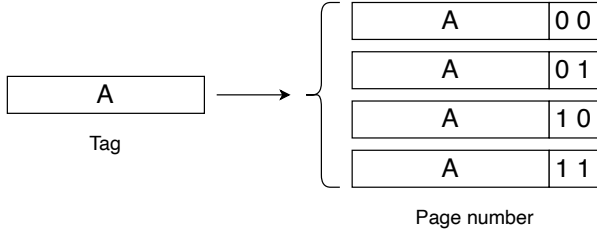
I. Tag vs page number deduplication

Seznec [26] observed that page numbers are redundantly stored across multiple front-end structures (e.g., L1-I cache, TLB, and BTB), appearing in fields such as cache tags, TLB tags, BTB tags, and BTB targets, etc. To eliminate this cross-structure duplication, he proposed storing each page number once and referencing it via pointers. For virtual page numbers, the proposal reuses the instruction TLB (I-TLB) rather than introducing a new structure.

While Seznec’s approach targets cross-structure duplication, LiteBTB focuses on eliminating duplication *within* a single structure, namely the BTB. Although Seznec’s technique



(a) the tag is two bits narrower than the page number.



(b) one tag maps to four page numbers.

Fig. 13: An example of tag and page number sizes for an 8-way 32K-entry BTB and 4KB pages.

can be adapted for intra-BTB deduplication, it differs from LiteBTB in two key aspects. First, LiteBTB deduplicates BTB tags, whereas Seznec’s approach deduplicates page numbers. These are not equivalent: tag width depends on BTB capacity and associativity, and may be either narrower or wider than the page number. Second, LiteBTB introduces a dedicated structure, the unique tag buffer (UTB), to store tags, whereas Seznec’s design reuses the I-TLB. We observe that reusing I-TLB (and not a dedicated structure) for deduplicating page numbers (and not BTB tags) has three fundamental limitations. Below we describe these limitations in detail.

1. Deduplicating page numbers rather than tags leads to an inefficient design: Page numbers and BTB tags are not equally sized. Page number width is determined by the page size, while tag width is dictated by the BTB capacity and associativity.

At large BTB capacities, tag widths become smaller than page numbers, causing a single tag to correspond to multiple distinct page numbers. Consequently, the I-TLB may store multiple page numbers that map to the same tag, effectively duplicating tag information. For example, as shown in Figure 13a, in a 48-bit virtual address space with 4KB pages, the page number spans bits 47:12 (36 bits), while an 8-way, 32K-entry BTB requires only 34 bits (bits 47:14, excluding the two least significant zero bits in ARM ISA) for a full tag. Because bits 13:12 are included in the page number but not in the tag, each tag can correspond to up to four page numbers (Figure 13b), all of which may co-reside in the I-TLB. This redundancy reduces the effective capacity of the I-

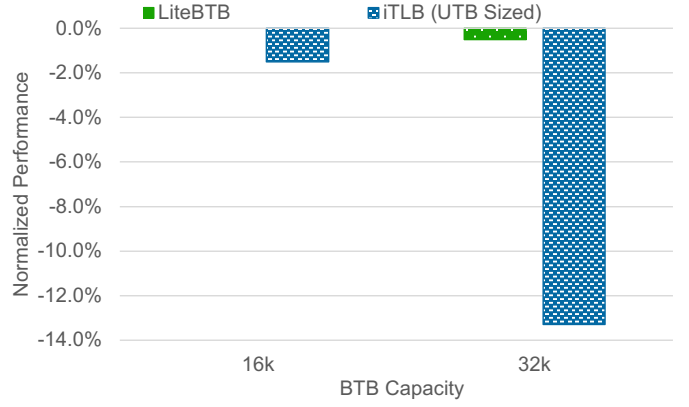


Fig. 14: Performance of LiteBTB and I-TLB based design of Seznec normalized to a baseline BTB-X without tag deduplication. LiteBTB and I-TLB are sized equally, so the performance difference is from using tag (in LiteBTB) vs page numbers (in I-TLB).

TLB for unique tags, and the effect worsens as BTB capacity increases and tag widths shrink. Consequently, the ongoing trends toward larger BTBs make page-number deduplication increasingly unattractive. LiteBTB avoids this limitation by deduplicating full BTB tags, rather than page numbers.

We quantify the performance impact of such tag duplication in I-TLB and how it fares against LiteBTB. For this study, we focus on 16K- and 32K-entry BTBs, where tag widths are 1 bit and 2 bits smaller than the page number, respectively. For a fair comparison, we size the I-TLB to match the UTB capacities listed in Table V, i.e. 768 and 512 entries for 16K- and 32K-entry BTB respectively. The results presented in Figure 14 show that the I-TLB approach incurs a performance loss of nearly 2% for the 16K-entry BTB and 13% for the 32K-entry BTB, whereas LiteBTB degrades performance by only about 0.5% relative to a baseline BTB-X without deduplication. The higher penalty for 32K-entry BTB is due to the larger difference between the tag and page number sizes - the tag is two bits narrower, thus each tag maps to four page numbers (versus two in the 16K BTB), increasing tag duplication in the I-TLB. These results demonstrate that deduplicating full BTB tags - rather than just page numbers - becomes increasingly important as BTB capacities grow.

Conversely, when tag width exceeds page number width, the BTB must store the remaining tag bits explicitly, limiting deduplication benefits because the entire tag is not deduplicated. For example, as shown in Figure 15, an 8-way, 2K-entry BTB requires a 38-bit tag (bits 47:10), while the page number requires 36 bits. The additional two tag bits (bits 11:10) are not stored in the I-TLB and therefore must be stored in the BTB. One could reduce BTB associativity to increase the number of index bits and shrink the tag, but this increases conflict misses. LiteBTB avoids these trade-offs by storing the complete BTB tag in the UTB, enabling full tag deduplication without modifying BTB associativity.

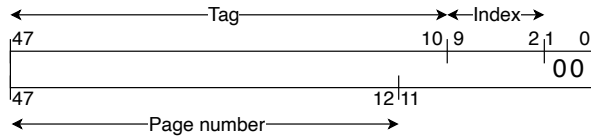


Fig. 15: An example of tag and page number sizes for an 8-way 2K-entry BTB and 4KB pages: the tag is two bits wider than the page number. Hence, the complete tag cannot be stored in the I-TLB.

2. Limited I-TLB capacity leads to frequent BTB misses and performance loss: In modern CPUs, the I-TLB and BTB are sized independently, and BTB capacities have grown so large that the I-TLB can store page numbers for only a fraction of the branches tracked by the BTB. Our analysis (Table V) shows that, depending on BTB size, up to 768 UTB entries are needed to hold the unique tags, whereas I-TLB capacity in modern CPUs is typically limited to only 128 to 256 entries. This capacity mismatch means that many BTB-resident branches do not find their corresponding page number in the I-TLB, leading to a substantial number of BTB misses and degraded performance. LiteBTB avoids this problem because its UTB is a dedicated structure whose capacity can be scaled directly with BTB capacity, ensuring it can accommodate the necessary unique tags.

To quantify the impact of limited I-TLB capacity, we model typical I-TLB capacities found in modern processors, i.e. 128 and 256 entries. The results presented in Figure 16 show that even with 256 entries, the I-TLB approach results in significant performance loss ranging from 5% to nearly 20% compared to only about 0.5% of LiteBTB over the baseline BTB-X design without any tag deduplication. This result emphasizes that I-TLB capacity in modern CPUs is not sufficient to hold the page numbers of branches present in BTB, thus resulting in a large performance penalty for I-TLB based approach and motivating the need for a dedicated storage for unique tags.

3. Stale pointers due to Page promotions and demotions: Modern I-TLBs support multiple page sizes, including standard 4KB pages and larger huge pages (e.g., 2MB/1GB). The operating system dynamically promotes multiple standard pages to a huge page or demotes a huge page back to standard pages. Consequently, I-TLB pointers stored in the BTB can become stale on page promotions and demotions. Handling this staleness requires either updating the pointers, thus introducing design complexity, or tolerating stale pointers, which causes increased BTB misses and degrades performance. LiteBTB avoids this issue entirely, as its UTB design is independent of page size.

VI. RELATED WORK

The core front-end is a well-known performance bottleneck in modern workloads, motivating extensive research on improving structures such as the BTB [5]–[7], [12]–[15], [27]–[32], instruction cache (L1-I) [1], [33]–[44], and branch

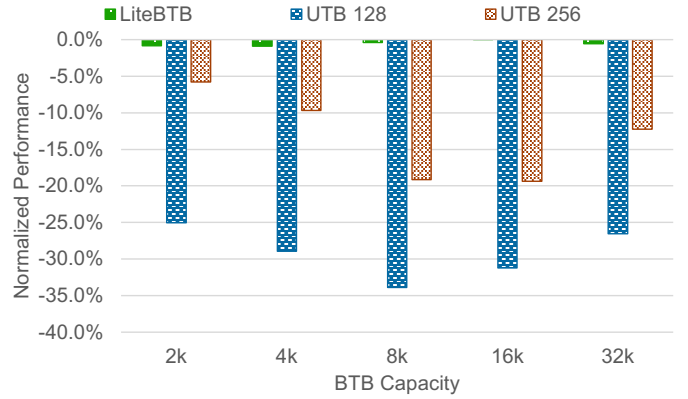


Fig. 16: Performance of LiteBTB and I-TLB based design with realistically sized 128- and 256-entry I-TLBs over a baseline BTB without tag deduplication.

predictors [45]–[50]. Among these, the BTB plays a central role as it lies on the critical path for instruction delivery and instruction prefetching. Prior work has explored multiple aspects of BTB design, including storage-efficient organization, tag reduction, prefetching, and replacement policies.

A significant body of work focuses on reducing BTB storage by optimizing branch target representation. Seznec et al. [51] observed that targets within the same page share a common page number and proposed storing page numbers in a separate structure while keeping only a pointer and offset in the BTB. Hoogerbrugge [52] introduced two differently sized entries in BTB to efficiently encode small target offsets. PDede [13] combines these ideas by distinguishing same-page and inter-page branches, eliminating redundant page storage for the former and using pointer-based representations for the latter. MicroBTB [14] further improves storage efficiency by adaptively packing one or two targets per entry depending on offset size. More recently, BTB-X [12] leverages the skewed distribution of branch offsets and employs asymmetric entry sizing to achieve dense packing and state-of-the-art storage efficiency. While these designs substantially reduce target storage, they leave tag storage largely unoptimized. In contrast, this work observes that tags now dominate BTB storage and targets this emerging bottleneck.

Prior work has also explored reducing tag storage through truncation and hashing. Early work by Fagin [53] demonstrated that BTBs can operate effectively with partial tags, achieving near-full performance with only a few bits. Modern processors employ techniques such as truncation and folded-XOR [16] to reduce tag width. Seznec’s DUPN [26] also enables deduplication of page numbers across front-end structures by reusing the I-TLB. While effective for cross-structure redundancy, this approach differs fundamentally from LiteBTB, which targets intra-BTB redundancy and operates directly on BTB tags rather than page numbers.

In addition to structural optimizations, several works aim to mitigate BTB misses through prefetching. Profile-guided techniques such as Twig [5] insert software prefetch instructions to

populate BTB entries ahead of demand. Hardware approaches such as Phantom BTB [54] virtualize BTB state in the cache hierarchy, while Boomerang [35] and Shotgun [36], [55] enhance fetch-directed instruction prefetching by improving BTB coverage. Prior work has also explored BTB replacement policies [7], [56] and hybrid designs such as AVM-BTB [6], which share storage between the BTB and instruction cache. These techniques are largely orthogonal to BTB organization and can be combined with the approach proposed in this work.

More broadly, instruction prefetching has been widely studied to address front-end bottlenecks. Temporal stream prefetchers [33], [34], [57] provide high accuracy but incur significant metadata overhead. Consequently, recent work focuses on fetch-directed instruction prefetching (FDIP) [1] and its extensions [35]–[38], which achieve strong performance with modest storage cost. Storage-efficient instruction cache designs, such as UBS [58], further aim to reduce unused capacity. These approaches are complementary to this work, as their effectiveness depends on accurate and timely BTB predictions.

VII. CONCLUSION

This work revisits the design of modern BTBs in the context of aggressively optimized branch target representations and identifies a key shift in the storage bottleneck: tags, rather than targets, now dominate BTB storage. We further show that BTB tags exhibit substantial redundancy, which increases with BTB capacity and presents a new opportunity for improving storage efficiency.

To exploit this redundancy, we propose LiteBTB, a storage-efficient BTB design that deduplicates tags by storing each unique tag once in a centralized Unique Tag Buffer (UTB) and replacing per-entry tags with compact pointers. LiteBTB preserves access latency by enabling parallel UTB and BTB lookups and mitigates aliasing through an efficient reactive scrubbing mechanism that avoids the complexity of proactive approaches. Our evaluation demonstrates that LiteBTB reduces BTB storage by 5.4–13.1% compared to the state-of-the-art BTB-X while maintaining comparable performance.

ACKNOWLEDGMENT

We thank the reviewers for their valuable feedback. This work is partially supported through the Research Council of Norway (NFR) grant 302279 to NTNU.

REFERENCES

- [1] G. Reinman, B. Calder, and T. Austin, “Fetch directed instruction prefetching,” in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 16–27.
- [2] A. Pellegrini, N. Stephens, M. Bruce, Y. Ishii, J. Pusdesris, A. Raja, C. Abernathy, J. Koppanalil, T. Ringe, A. Tummala, J. Jalal, M. Werkheiser, and A. Kona, “The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc,” *IEEE Micro*, vol. 40, no. 2, pp. 53–62, 2020.
- [3] A. Saporito, “The IBM z15 processor chip set,” in *Hot Chips*, 2020.
- [4] D. Suggs, M. Subramony, and D. Bouvier, “The AMD “zen 2” processor,” *IEEE Micro*, vol. 40, no. 2, pp. 45–52, 2020.
- [5] T. A. Khan, N. Brown, A. Sriraman, N. K. Soundararajan, R. Kumar, J. Devietti, S. Subramoney, G. A. Pokam, H. Litz, and B. Kasikci, “Twig: Profile-guided btb prefetching for data center applications,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 816–829. [Online]. Available: <https://doi.org/10.1145/3466752.3480124>
- [6] Y. Liu, X. Li, T. Zhang, T. Liu, Q. Guo, F. Zhang, and J. Wang, “Avm-btb: Adaptive and virtualized multi-level branch target buffer,” in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ser. ISCA '24. IEEE Press, 2025, p. 17–31. [Online]. Available: <https://doi.org/10.1109/ISCA59077.2024.00012>
- [7] S. Song, T. A. Khan, S. Mahdizadeh Shahri, A. Sriraman, N. K. Soundararajan, S. Subramoney, D. A. Jiménez, H. Litz, and B. Kasikci, “Thermometer: Profile-guided btb replacement for data center applications,” in *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, ser. ISCA 2022, Jun. 2022.
- [8] “IBM z16 (3931) technical guide. Section 3.4.3, page 84,” <https://www.redbooks.ibm.com/redbooks/pdfs/sg248951.pdf>.
- [9] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito, “The ibm z15 high frequency mainframe branch predictor,” in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20. IEEE Press, 2020, p. 27–39. [Online]. Available: <https://doi.org/10.1109/ISCA45697.2020.00014>
- [10] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, “Evolution of the samsung exynos cpu microarchitecture,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 40–51.
- [11] L. Eeckhout, “Focal: A first-order carbon model to assess processor sustainability,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 401–415. [Online]. Available: <https://doi.org/10.1145/3620665.3640415>
- [12] T. Asheim, B. Grot, and R. Kumar, “A storage-effective btb organization for servers,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 1153–1167.
- [13] N. K. Soundararajan, P. Braun, T. A. Khan, B. Kasikci, H. Litz, and S. Subramoney, “Pdede: Partitioned, deduplicated, delta branch target buffer,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 779–791. [Online]. Available: <https://doi.org/10.1145/3466752.3480046>
- [14] V. Gupta and B. Panda, “Micro btb: A high performance and storage efficient last-level branch target buffer for servers,” in *Proceedings of the 19th ACM International Conference on Computing Frontiers*, ser. Cf '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 12–20. [Online]. Available: <https://doi.org/10.1145/3528416.3530224>
- [15] T. Asheim, B. Grot, and R. Kumar, “BTB-X: A storage-effective BTB organization,” *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 134–137, 2021.
- [16] L. Li, H. Yavarzadeh, and D. Tullsen, “Indirector: High-Precision branch target injection attacks exploiting the indirect branch predictor,” in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2137–2154. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity24/presentation/li-luyi>
- [17] “ChampSim Simulator,” <https://github.com/ChampSim/ChampSim>.
- [18] 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.
- [19] “1st Instruction Prefetching Championship,” <https://research.ece.ncsu.edu/ipc/>.
- [20] J. Feliu, A. Perais, D. A. Jiménez, and A. Ros, “Rebasing microarchitectural research with industry traces,” in *2023 IEEE International Symposium on Workload Characterization (IISWC)*, 2023, pp. 100–114.
- [21] C. Pepi, K. Tibrewala, B. R. Godala, S. P. Ramesh, A. Ros, D. A. Jiménez, G. A. Pokam, and P. V. Gratz, “Correct wrong path simulation,” in *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2026, <https://webs.um.es/aros/papers/pdfs/cpepi-ispass26.pdf>.
- [22] “GitHub - google/cityhash: Automatically exported from code.google.com/p/cityhash — code.google.com,” <https://code.google.com/p/cityhash/>, [Accessed 17-06-2026].

- [23] M. O. Rabin, "Fingerprinting by random polynomials," *Technical report*, 1981.
- [24] M. J. Dworkin *et al.*, "Sha-3 standard: Permutation-based hash and extendable-output functions," 2015.
- [25] A. L. Zobrist, "A new hashing method with application for game playing," *ICGA Journal*, vol. 13, no. 2, pp. 69–73, 1990.
- [26] A. Seznec, "Don't use the page number, but a pointer to it," in *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ser. Isca '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 104–113. [Online]. Available: <https://doi.org/10.1145/232973.232985>
- [27] T. Asheim, B. Grot, and R. Kumar, "A specialized BTB organization for servers," in *Proceedings of the 31st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2022.
- [28] A. Perais and R. Sheikh, "Branch target buffer organizations," in *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023, pp. 240–253.
- [29] S. Mirbagher Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 519–532.
- [30] C. Pepi, B. R. Godala, K. Tibrewala, G. A. Chacon, P. V. Gratz, D. A. Jiménez, G. A. Pokam, and D. I. August, "Skia: Exposing shadow branches," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1091–1106. [Online]. Available: <https://doi.org/10.1145/3676641.3716273>
- [31] R. K. Brunner and R. Kumar, "Driving the core frontend with LiteBTB," *IEEE Computer Architecture Letters*, vol. 25, no. 1, pp. 57–60, 2026.
- [32] R. K. Brunner and R. Kumar, "Understanding btb tag sizing," in *2026 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2026, pp. 369–379.
- [33] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.
- [34] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared History Instruction Fetch for Lean-core Server Processors," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.
- [35] R. Kumar, C.-C. Huang, B. Grot, and V. Nagarajan, "Boomerang: A metadata-free architecture for control flow delivery," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 493–504.
- [36] R. Kumar, B. Grot, and V. Nagarajan, "Blasting through the front-end bottleneck with shotgun," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. Asplos '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 30–42. [Online]. Available: <https://doi.org/10.1145/3173162.3173178>
- [37] S. Oh, M. Xu, T. A. Khan, B. Kasikci, and H. Litz, "Udp: Utility-driven fetch directed instruction prefetching," in *51st International Symposium on Computer Architecture (ISCA)*. ISCA, 2024.
- [38] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Seznec, D. Tullsen, and D. I. August, "Pdip: Priority directed instruction prefetching," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 846–861. [Online]. Available: <https://doi.org/10.1145/3620665.3640394>
- [39] A. Ros and A. Jimborean, "A cost-effective entangling prefetcher for instructions," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 99–111. [Online]. Available: <https://doi.org/10.1109/ISCA52012.2021.00017>
- [40] 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*, 2019, pp. 462–473.
- [41] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-SPY: Context-driven conditional instruction prefetching with coalescing," vol. 2020-October. IEEE Computer Society, 10 2020, pp. 146–159.
- [42] S. Jamilan, T. A. Khan, G. Ayers, B. Kasikci, and H. Litz, "Apt-get: profile-guided timely software prefetching," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 747–764. [Online]. Available: <https://doi.org/10.1145/3492321.3519583>
- [43] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: profile-guided instruction cache replacement for data center applications," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 734–747. [Online]. Available: <https://doi.org/10.1109/ISCA52012.2021.00063>
- [44] A. V. Jamet, G. Vavouliotis, M. Torrents, D. Chasapis, and M. Casas, "Enhancing instruction prefetching via cache and tlb management," in *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*, 2026, pp. 2254–2269.
- [45] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *Journal of Instruction-Level Parallelism (JILP)*, vol. 8, 2006.
- [46] D. Jimenez and C. Lin, "Dynamic branch prediction with perceptrons," in *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, 2001, pp. 197–206.
- [47] 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 International Symposium on Microarchitecture (MICRO)*, ser. MICRO 2022, Oct. 2022.
- [48] D. Schall, A. Sandberg, and B. Grot, "The last-level branch predictor," in *Proceedings of the 57th Annual IEEE/ACM International Symposium on Microarchitecture*, 2024.
- [49] D. Schall, M. Ďuračková, and B. Grot, "The last-level branch predictor revisited," in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2026, pp. 1–16.
- [50] S. Zangeneh, S. Pruett, S. Lym, and Y. N. Patt, "Branchnet: A convolutional neural network to predict hard-to-predict branches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 118–130.
- [51] A. Seznec, "A 64-kbytes ittag indirect branch predictor," *J. Instruction-Level Parallelism*, 2011.
- [52] J. Hoogerbrugge, "Cost-efficient branch target buffers," in *Euro-Par 2000 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 950–959.
- [53] B. Fagin and K. Russell, "Partial resolution in branch target buffers," in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 193–198.
- [54] 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, 2009*, pp. 313–324. [Online]. Available: <http://doi.acm.org/10.1145/1508244.1508281>
- [55] R. Kumar and B. Grot, "Shooting down the server front-end bottleneck," *ACM Trans. Comput. Syst.*, vol. 38, no. 3–4, jan 2022. [Online]. Available: <https://doi.org/10.1145/3484492>
- [56] S. Mirbagher Ajorpaz, E. Garza, S. Jindal, and D. A. Jiménez, "Exploring predictive replacement policies for instruction cache and branch target buffer," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 519–532.
- [57] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: Unified instruction supply for scale-out servers," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 166–177.
- [58] R. K. Brunner and R. Kumar, "Weeding out front-end stalls with uneven block size instruction cache," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 1382–1396.