Driving the Core Frontend with LiteBTB

Roman K. Brunner and Rakesh Kumar

***Abstract*—Branch target buffer (BTB) is a central component of high performance core front-ends as it not only steers instruction fetch by uncovering upcoming control flow but also enables highly effective fetch-directed instruction prefetching. However, the massive instruction footprints of modern server applications far exceed the capacities of moderately sized BTBs, resulting in frequent misses that inevitably hurt performance. While commercial CPUs deploy large BTBs to mitigate this problem, they incur high storage and area overheads. Prior efforts to reduce BTB storage have primarily targeted branch targets, which has proven highly effective — so much that the tag storage now dominates the BTB storage budget. We make a key observation that BTBs exhibit a large degree of tag redundancy, i.e. only a small fraction of entries contain unique tags, and this fraction falls sharply as BTB capacity grows. Leveraging this insight, we propose LiteBTB, which employs a dedicated hardware structure to store unique tags only once and replaces per-entry tags in BTB with compact tag pointers. To avoid latency overheads, LiteBTB accesses the tag storage and BTB in parallel. Our evaluation shows that LiteBTB reduces storage by up to 13.1% compared to the state-of-the-art BTB design, called BTB-X, while maintaining equivalent performance. Alternatively, with the same storage budget, LiteBTB accommodates up to 1.125× more branches, yielding up to 2.7% performance improvement.**

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 (L1I), 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) which is commonly employed in commercial CPUs [6], [8]. Indeed, recent research has shown that an ideal BTB provides significantly higher performance benefits than an ideal L1I [5].

To mitigate the performance cost of BTB misses, commercial processors feature massive BTBs, with capacities growing substantially across successive 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 [1] — over 10× larger than the 24K-entry BTB in zEC12 [2] 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 [4].

Although these enormous BTBs are very effective in minimizing BTB misses, they incur large storage and area overheads. Further, the slowing down of Moore's law makes the trend of increasingly larger BTBs unsustainable. Therefore, it is critical to investigate techniques for reducing BTB storage overhead without hurting their hit rates. We observe that at the current BTB sizes, saving even a single 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.

Authors are with Norwegian University of Science and Technology (NTNU), Norway.

V	Tag	Type	Repl-Policy	Target
1b	13-15b	2b	3b	46b

Fig. 1: Composition of a conventional BTB entry.

Branch targets dominate storage requirements in a conventional BTB design, as shown in Figure 1, irrespective of whether the BTB is organized in an instruction-, block-, or region-based manner [7]. Therefore, recent work [3] has focused on optimizing target representation to control the growing storage and area costs of BTBs. Indeed, the branch target representation in the recently proposed BTB designs is so optimized that now tags account for the bulk of BTB storage requirements rather than branch targets. Specifically, in BTB-X, the state-of-the-art BTB design, tags consume 43% of the storage budget, while branch targets consume only 35%.

This work seeks to reduce tag storage overhead to maximize the number of BTB entries in a given storage budget. To that end, we make the key observation that the number of unique tags present in the BTB is significantly smaller than the number of 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 needs.

Building on this insight, we propose to store each unique tag only once in a dedicated hardware structure and store a pointer to the tag in the BTB. Since the number of unique tags is small, the pointers require fewer bits than the tags, yielding substantial storage savings. However, a naive design for de-duplicating the tags might increase BTB access latency: the BTB must first be accessed to fetch the pointer, followed by a second access to retrieve the tag itself. This serialization of tag pointer retrieval with subsequent tag read results in higher access latency compared to a conventional design that stores tags directly in the BTB. This work introduces LiteBTB, a BTB design that eliminates this serialization by orchestrating the dedicated tag storage such that it can be accessed in parallel with the BTB accesses.

Our evaluation shows that LiteBTB needs 2.6%-13.1% less storage than BTB-X for 2K-32K branches while delivering similar performance. Conversely, at a fixed storage budget, it tracks 1.125× more branches, yielding up to 2.7% speedup.

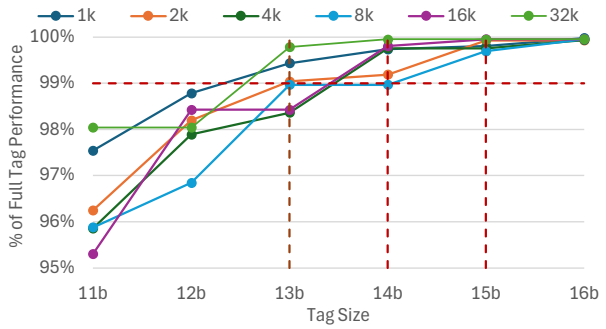

Fig. 2: Performance at different tag sizes for different BTB capacities, normalized to full tag performance.

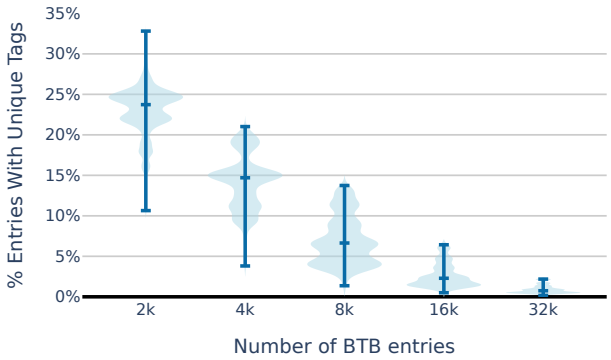

Fig. 3: Fraction of BTB entries containing unique tags.

II. Motivation

Sizing BTB tags: The size of tags offers a trade-off between BTB storage requirements and aliasing. A full tag, i.e. all the program counter (PC) bits not used for indexing the BTB, precisely identifies whether the indexed BTB entry corresponds to the PC; however, it incurs high storage overhead. In contrast, a partial tag, generated by dropping some of the most significant tag bits, reduces storage requirements. However, it also introduces aliasing, i.e. false hits, as a mismatch in the dropped tag bits cannot be detected. This aliasing takes the execution down the wrong path, which eventually results in pipeline flush and performance loss.

We did a study to understand the impact of tag size on performance and the results are presented in Figure 2. As the figure shows, a 16-bit tag provides the same performance as the full tag for all BTB sizes. However, reducing the tag size further starts to hurt performance progressively. Therefore, we size the tags in our baseline BTB design, i.e. BTB-X, to achieve at least 99% of the performance of the full tag. Accordingly, the tag sizes are set to 13, 14, 15, 14, and 13 bits for 2K-, 4K-, 8K-, 16K-, and 32K-entry BTBs, respectively.

Analyzing unique tags: 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 $2^{12} = 4$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 tag presents in differently sized BTBs. It 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 and, 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.

III. LiteBTB Design

To reduce the tag storage overhead, we leverage the insight gained in the previous section that there is a large degree of tag duplication in BTBs. Based on this insight, we propose to deduplicate the tags by storing each unique tag only once. We can store the unique tags in a dedicated hardware structure such that a BTB entry stores only a pointer to the required tag. Given that a very small number of unique tags need to be tracked, the pointer size will be smaller than the tag size itself, thus resulting in lower storage requirements. Although such a design reduces storage requirements, it might increase the BTB access latency depending on how the dedicated unique tag storage is organized. Therefore, we first discuss a naive approach before presenting the proposed LiteBTB design.

A. A naive design

A naive design would introduce additional latency to BTB hit detection, thus increasing overall BTB access time, by serializing accesses to the BTB and tag storage. Since pointers are stored in the BTB, such a design requires a BTB access to read the pointer followed by a separate access to retrieve the corresponding tag from the tag storage. This serialization increases tag access latency compared to a conventional BTB, where tags are stored in the BTB itself and can be read in a single access. The resulting increase in overall BTB access time degrades performance, making this design unattractive.

B. LiteBTB

The key idea behind our proposed BTB design, LiteBTB (Figure 4), is to access the tag storage—referred to as the unique tag buffer (UTB)—in parallel with the BTB. For that, unlike a naive design that reads the *tag* from the tag storage, LiteBTB retrieves the *tag location* from UTB. Specifically, UTB is accessed using the tag bits from the current PC rather than with the tag pointer stored in BTB. This allows UTB and BTB to be accessed in parallel, as the tag and index bits required for accessing these structures can be extracted from the PC simultaneously. The tag's location retrieved from the UTB is then compared against the pointer from the BTB, with a match indicating a BTB hit.

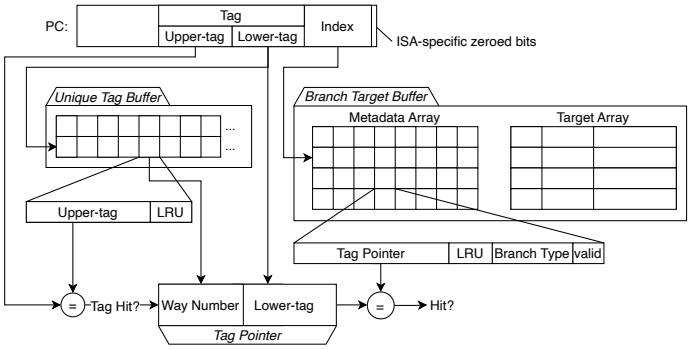

Fig. 4: LiteBTB microarchitecture

UTB organization: For maximum flexibility in tag placement, UTB could be implemented as a fully associative structure. However, the complexity of fully associative designs, due to the number of comparisons required to check for tag presence, grows rapidly beyond a few tens of entries. To address this, we organize UTB as a set-associative structure. Another disadvantage of fully associative UTB is that it would require storing all tag bits. 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 UTB are accessed using the tag and index bits extracted from the PC. As BTB is typically a set-associative structure, the tag pointers are read from all ways of the indexed set. Simultaneously, the lower-tag bits are used to index UTB, and the upper-tag bits stored in all ways of the indexed set are retrieved and compared with the upper-tag bits of the current PC. If no match is found, then the tag is not present in UTB and the access is considered a miss. Otherwise, the lower-tag bits are concatenated with the way number of the matching entry to form the tag pointer, which is then compared against the pointers read from the indexed BTB set. A match at this stage indicates 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 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 corresponding BTB entry (or entries) is not invalidated, as maintaining such a mapping from UTB to BTB entries would incur substantial storage and design complexity overheads. Consequently, these entries will experience a miss on their next access, since their corresponding tag will no longer be present in the UTB.

IV. Evaluation

Methodology: We use ChampSim, an open-source trace-driven simulator, to evaluate the efficacy of LiteBTB on server, client, and SPEC CPU workload traces provided by Qualcomm for the first Instruction Prefetching Championship (IPC-1). We warm up microarchitectural structures for 50M instructions and collect statistics over the next 50M. We use BTB-X [3] as our baseline BTB. The microarchitectural parameters for the modeled processor are listed in Table I.

TABLE I: 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	Hashed Perceptron
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, 36-27 cycles latency, 64 MSHRs

TABLE II: BTB-X and LiteBTB storage requirements

BTB Entries	BTB-X	LiteBTB			**Savings**
	Size [KB]	UTB Entries	UTB Size [KB]	BTB Size [KB]	
2K	7.25	512	0.813	7.063	**2.59%**
4K	15.00	768	1.219	14.219	**5.21%**
8K	31.00	832	1.422	27.422	**11.54%**
16K	60.00	768	1.219	53.219	**11.30%**
32K	116.00	512	0.813	100.813	**13.09%**

Storage requirements: The storage requirements of the baseline BTB-X and LiteBTB across different BTB capacities are reported in Table II. Both designs are 8-way set associative, and BTB capacity is doubled by doubling the number of sets.

To determine the right balance between UTB and BTB capacities, we fix BTB capacity and sweeps UTB capacity to find the smallest UTB that nearly matches baseline performance. As shown in Figure 5, the optimal UTB size depends on BTB capacity: 2K and 32K BTBs need 512 entries, 4K and 16K need 768, and 8K requires 832 UTB entries to reach 99% of baseline performance. Notice that the UTB capacity increases as BTB size grows from 2K to 8K entries, but decreases beyond that point. This trend is governed by two opposing factors. First, larger BTBs incur fewer evictions, thus they are likely to hold more unique tags. Second, as defined in Section II, the region size grows with BTB capacity, and larger regions cause more PCs to collapse to a common tag, thereby reducing the number of unique tags in the BTB. The first factor dominates up to the 8K-entry BTB configuration, after which the second factor becomes dominant.

Table II highlights that LiteBTB achieves increasingly higher storage savings over BTB-X as BTB capacity scales. For 2K- and 4K-entry organizations, LiteBTB reduces storage by 2.5% and 5.2%, respectively. At larger capacities with 8K, 16K, and 32K entries, the savings grow to 11.5%, 11.3%, and 13.1%. This is a significant result given the steady growth of BTB capacity in modern processors.

The storage efficiency of LiteBTB arises from the smaller size of tag pointers compared to tags in BTB-X. Specifically, tag pointers are four bits shorter than tags for 2K-, 4K-, 16K-, and 32K-entry BTBs, and five bits shorter for the 8K-entry configuration. Moreover, UTB storage constitutes only a negligible fraction of total storage, particularly at larger capacities. For example, UTB accounts for less than 1% of the total storage in a 32K-entry LiteBTB. This is because UTB capacity is very small compared to the BTB, and each entry contains only a handful of bits.

Performance: Figure 6 compares LiteBTB and BTB-X under two scenarios: 1) equal number of entries, and 2) equal storage

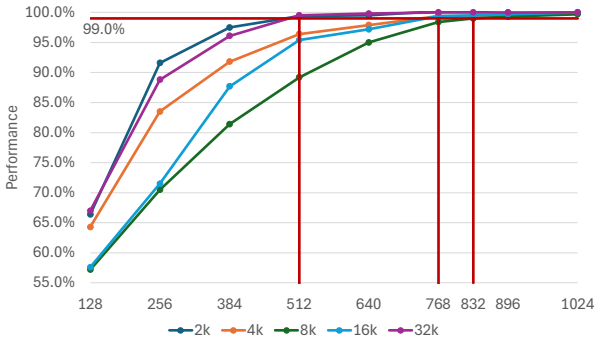

Fig. 5: LiteBTB speedup over BTB-X across UTB-BTB sizes.

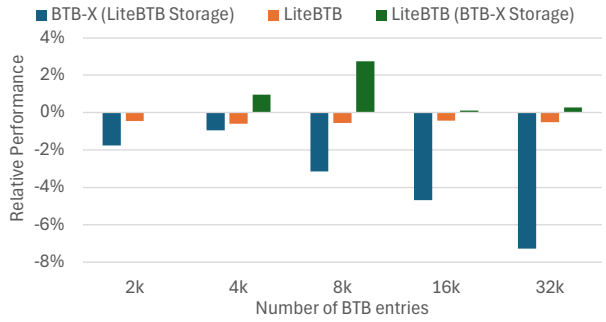

Fig. 6: Speedup of different BTB designs over BTB-X.

budget. With equal entries, LiteBTB achieves the storage savings reported in Table II, whereas with equal storage it delivers better performance than BTB-X. For the equal-storage comparison, we evaluate two configurations. First, BTB-X is constrained to the same storage budget as required by LiteBTB in Table II. To meet this budget, the tag size in BTB-X is reduced, consistent with the tag-sizing focus of this work. This configuration is denoted *BTB-X (LiteBTB Storage)* in Figure 6. Second, LiteBTB is allocated the same storage as BTB-X in Table II. Since the tag sizes used in this study already achieve 99% of the performance of a full tag, increasing the tag size will bring minimal performance. Instead, the additional storage is utilized to provision an additional way in the BTB. This configuration is denoted *LiteBTB (BTB-X Storage)* in Figure 6.

Figure 6 presents the performance gain of *BTB-X (LiteBTB Storage)*, *LiteBTB*, and *LiteBTB (BTB-X Storage)* over BTB-X. LiteBTB incurs only about 0.5% or less performance loss across all BTB sizes, despite its lower storage budget. *BTB-X (LiteBTB Storage)*, in contrast, suffers significant performance degradation, especially at larger BTB sizes, exceeding 7% at 32K entries. This demonstrates that simply reducing the tag size to save storage results in substantial performance penalties. Finally, *LiteBTB (BTB-X Storage)* achieves 2.7% performance improvement at 8K entries due to the additional way. At 16K and 32K entries, however, the gain diminishes because the branch working set starts to fit in the BTB and the one additional way does not increase the hit rate substantially.

Latency analysis: Although the UTB and BTB are accessed in parallel, LiteBTB's latency could increase if UTB cannot supply the tag location before the tag pointers are read out from BTB for comparison. Using CACTI 7.0 at 22 nm, we evaluate this for an 8K-entry LiteBTB. CACTI reports a 0.16ns BTB tag pointer read latency, while the UTB returns the tag location in 0.14ns, indicating that the UTB does not extend the critical path. LiteBTB's overall tag hit latency is 0.295ns compared to 0.339ns for BTB-X, primarily because it compares smaller tag pointers. Further, Table III shows that LiteBTB also reduces area by 10.3% and per-access read and write energy by 12.5% and 17.6%, respectively, over BTB-X.

TABLE III: Area and Energy Impact of LiteBTB

		BTB-X	LiteBTB		**Savings**
			UTB	BTB	
Area [mm^2]		0.376	0.020	0.317	**10.3%**
Energy per Access [nJ]	read	0.032	0.003	0.025	**12.5%**
	write	0.034	0.008	0.020	**17.6%**

Comparison with prior work: Seznec [9] also reduces BTB tag storage by deduplicating page numbers via the I-TLB, but LiteBTB differs in two key aspects. First, LiteBTB deduplicates BTB tags, whereas [9] deduplicates page numbers. At large BTB capacities, tags become narrower than page numbers, so a single tag maps to multiple page numbers, effectively causing redundant tags in I-TLB which reduces its effective capacity. Consequently, with equal UTB and I-TLB capacities, LiteBTB outperforms [9] by 14% for a 32K-entry BTB. Conversely, for small BTBs, tags exceed page number width, forcing the BTB to store the tag bits that I-TLB cannot deduplicate, limiting its benefit. Second, LiteBTB uses a dedicated unique tag buffer (UTB), while [9] relies on the I-TLB. Table II shows we need 512–832 UTB entries depending on BTB size, whereas modern I-TLBs typically provide only 128–256 entries. Due to this limited I-TLB capacity, many BTB-resident branches do not find their page number in the I-TLB, leading to additional BTB misses and performance loss.

V. Conclusion

Commercial CPUs incur high storage and area costs to deploy large BTBs that reduce miss-induced performance loss. With prior optimizations improving branch-target representation, tag storage now dominates BTB storage budget. We observe that duplicate tags are a major contributor. LiteBTB addresses this by storing each unique tag once in a dedicated structure and replacing per-entry tags with compact pointers, while accessing UTB and BTB in parallel to avoid added latency.

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] "IBMibm z16 (3931)technical guide. Section 3.4.3, page 84," https://www.redbooks.ibm.com/redbooks/pdfs/sg248951.pdf.
[2] N. Adiga *et al.*, "The IBM z15 high frequency mainframe branch predictor," in *ISCA*, 2020.
[3] T. Asheim *et al.*, "A storage-effective btb organization for servers," in *HPCA*, 2023.
[4] B. Grayson *et al.*, "Evolution of the Samsung Exynos CPU Microarchitecture," in *ISCA*, 2020.
[5] T. A. Khan *et al.*, "Twig: Profile-guided btb prefetching for data center applications," in *MICRO*, 2021.
[6] A. Pellegrini *et al.*, "The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc," *IEEE Micro*, 2020.
[7] A. Perais *et al.*, "Branch target buffer organizations," in *MICRO'23*.
[8] G. Reinman *et al.*, "Fetch directed instruction prefetching," in *MICRO*, 1999.
[9] S. Seznec, "Don't use the page number, but a pointer to it," in *ISCA'96*.