Understanding BTB Tag Sizing

Roman K. Brunner and Rakesh Kumar
Norwegian University of Science and Technology
Trondheim, Norway
{roman.k.brunner, rakesh.kumar}@ntnu.no

***Abstract*—The large branch footprints of contemporary applications easily overwhelm the capacity of Branch target buffers (BTBs). Therefore, to avoid frequent BTB misses and their associated performance penalties, commercial processors feature massive BTBs that require hundreds of KBs to multi-MB storage budgets. Furthermore, storage requirements are increasing at an alarming rate, a trend that is certainly not sustainable.**

As the bulk of BTB storage budget goes towards storing branch targets, researchers have recently proposed storage-efficient schemes for target representation. The state-of-the-art schemes are so effective that branch targets no longer dominate the BTB storage requirements, rather, the tags do. However, there has not been any study on understanding the implications of tag size on performance, storage, and aliasing. This work bridges this gap by performing a comprehensive study of how and why tag size requirements vary for performance- and storage-efficiency-oriented BTB designs across different BTB capacities. The results of the study help BTB designers understand where in BTB to invest any additional storage budget that becomes available in next processor generations.

The key findings of this study include: 1) moderately sized BTBs (2K to 8K entries) require larger tags than small (256-entry) and large (32K entry) BTBs, 2) increasing the number of ways in the BTB also requires larger tags to fully realize the benefits of the additional ways, and 3) a storage-efficient BTB design reduces tag storage requirements by up to 21% over a performance-oriented BTB design.

I. Introduction

Branch target buffer (BTB) is a cornerstone of modern high-performance core front-end designs as it guides instruction fetch by predicting the upcoming control flow and enables fetch directed instruction prefetching. Specifically, BTB steers instruction fetch across control flow discontinuities by locating branch instructions and providing the target address for predicted taken and unconditional branches. Further, it also enables fetch directed instruction prefetching (FDIP) [31], a commonly employed prefetching mechanism in commercial CPUs [29], [32], that leverages the upcoming control flow uncovered by the BTB to discover prefetching opportunities.

Nevertheless, the massive instruction footprints of modern server applications can easily overwhelm latency-sensitive front-end structures, including BTB, instruction cache (L1-I), etc. Consequently, the frequent BTB misses result in instruction fetch along wrong paths and subsequent pipeline flushes when the wrong path is detected in the core back-end. The pipeline flushes hurt performance as they throw away tens of cycles worth of work and also expose pipeline fill latency. Further, BTB misses also cause FDIP to prefetch along the wrong path, which pollutes the L1-I and on-chip network, thus further hurting performance. Indeed, prior work observed that BTB misses hurt performance even more than L1-I misses [22], [26], [35].

To avoid the performance cost of BTB misses, commercial CPUs across the computing spectrum, from servers to mobiles, deploy BTBs with enormous storage capacities. For example, the second-level BTB in the server-grade IBM z16 CPUs can accommodate up to a whopping 260K branches [1]. Furthermore, not only are BTB sizes huge, they are increasing at an unprecedented rate. Again taking IBM z-series as an example, the number of entries in the second-level BTB increased more than ten fold from 24K entries in zEC12 to 260K in z16, over merely 4 generations [6]. Similarly, on the other extreme, BTB storage in Samsung Exynos mobile CPUs saw nearly 6x increase from 90.9KB of storage in Exynos M1/M2 to 529KB in M6 [16].

Considering the continuously growing BTB storage requirements, researchers are investigating storage-efficient BTB organizations. Since branch targets occupy the bulk of the storage budget in a conventional BTB design (see Figure 1), prior work [4], [33], [36] has focused on optimizing the branch target representation. Seznec's design [33] and PDede [36] exploit an observation that all branch targets within a page share a common page number. Therefore, they store the page numbers, and region numbers in PDede, in a separate structure, and BTB stores only a pointer to it. A recent BTB design, called BTB-X [4], proposes storing branch target offsets rather than full target addresses, noting that this substitution yields substantial storage savings. To accommodate large variability in target offset sizes, BTB-X employs unevenly sized ways.

The prior work has been very effective in reducing the storage requirements of branch targets as PDede and BTB-X require about 49% and 35% of the BTB storage budget, respectively, for storing targets compared to 72% of a conventional BTB that stores full targets. In fact, BTB-X, the state-of-the-art BTB design, is so storage effective that it is the tags that now dominate its storage budget rather than the branch targets. Specifically, tags account for about 43% of the storage budget compared to targets requiring only 35%.

As tags start to constitute an increasingly large fraction of the BTB storage budget, it is critical to address this overhead to achieve an optimal storage-efficient design. However, there is no comprehensive prior study on understanding how tag sizing affects performance, BTB misses, and aliasing. The only prior work on this comes from three decades ago [11]. It proposes to store only a portion of the full tag in the BTB and concludes

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

Fig. 1: Composition of an entry in a conventional BTB with typical number of bits required for each field.

that a 4-bit tag is sufficient to achieve 99.9% accuracy of the full tag for all the BTB sizes and workloads evaluated. In contrast, both PDede and BTB-X use 12-bit tags.

As a step towards optimizing tag storage in modern BTBs, this work studies the impact of tag size on BTB performance, storage, and aliasing. Specifically, we examine the smallest tag size required for performance-oriented BTB designs, which aim to achieve a performance within 1% of the full tag, as well as for storage-efficient designs, which seek to maximize performance per KB of storage, across different BTB capacities. Furthermore, we explore how to distribute any additional storage budget, if provided to BTB designers, among increasing the number of entries and/or expanding the size of the tags. In summary, the key contributions of this work are as follows:

- Introduces and rigorously characterizes a previously unexplored problem - tag sizing in modern BTBs.
- Discovers that, for a fixed associativity, moderately-sized BTBs (2K-8K entries) need larger tags compared to both small (256-512 entries) and large (32K entries) BTBs. It implies that increasing the BTB size without adjusting the tag size results either in suboptimal performance or overprovisioned tag storage.
- Shows that increasing the number of *ways* in the BTB also requires larger tags to fully realize the benefits of the additional ways. Further, depending on BTB configuration, investing additional storage in larger tags might provide better performance than in additional ways.
- Reveals that, at small tag sizes, increasing BTB capacity by means of additional ways results in performance degradation instead of performance gain.
- Identifies that storage-efficient BTB designs require two to four fewer tag bits compared to performance-oriented designs. As a result, they reduce tag storage requirements by up to 21% and overall BTB storage by up to 14%; however, they achieve only 90%-97% of the full tag performance, as opposed to the 99% attained by the performance-oriented designs.

II. Background and Motivation

The branch prediction unit (BPU) plays a critical role in modern core front-end by predicting the upcoming control flow and steering the instruction fetch accordingly. In the absence of branch instructions, the control flow progresses sequentially, i.e. the instructions are fetched and executed successively one after the other in the same order as they are laid down by the compiler in memory. However, branches break this sequential execution by transferring the control flow to a non-contiguous instruction. Therefore, it is crucial to identify branches in order to redirect the fetch to their targets. However, whether or not an instruction is a branch can only be determined at the decode stage. To avoid the latency of instruction fetch and decode before generating the next PC, modern high performance front-ends employ a hardware structure called branch target buffer (BTB). The BTB is accessed with the program counter (PC), and a BTB hit indicates that the PC corresponds to a branch instruction.

CPU	L1 BTB	L2 BTB	Total
Samsung Exynos (storage)			
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	378.8KB
M6	78.5KB	451.0KB	529.5KB
IBM zSeries (entries)			
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 [1], [2], [16]

A. Branch Target Buffer (BTB)

BTB serves three key purposes. First, given a PC, it predicts whether the instruction at this PC is a branch or not. Second, for predicted branches, BTB provides the branch type, i.e. conditional, call, return, or unconditional jump. Finally, for predicted taken branches, BTB provides the branch target to which the fetch should be redirected. As such, BTB stores sufficient information for the front-end to speculatively identify branch instructions and redirect fetch to their targets without waiting for instructions to be fetched and decoded.

Figure 1 shows the composition of a conventional BTB entry along with the typical size of each field. Each entry includes *Valid*, *Tag*, *Type*, *Target*, and *Repl_policy* (replacement policy) fields. Optionally, it can also include information about direction prediction for conditional branches; however, modern BPUs usually incorporate a dedicated direction predictor.

The role of Tag field: BTB is indexed with the lower-order PC bits, and the tag field of the indexed entry is compared to the (subset of) remaining PC bits. A match signifies a BTB hit meaning that the PC corresponds to a branch instruction, and the indexed BTB entry contains information about the branch. The tag size offers a trade-off between BTB storage requirements and aliasing, i.e. false hits. Specifically, a full tag, i.e. all the PC bits that are not used to index BTB, precisely identifies if the indexed entry corresponds to the lookup PC or not; however, it requires a high storage budget. In contrast, a smaller tag, formed by omitting some bits from the full tag [11], reduces storage requirements; however, it introduces aliasing because a mismatch in the omitted bits goes undetected. Consequently, tag sizing must carefully balance storage efficiency, aliasing, and BTB capacity.

Component	Configuration
Core	6 wide fetch, decode and commit, 512 entry ROB, 205 entry scheduler, 192 entry load queue, 114 entry store queue
Branch Predictor	Hashed Perceptron
Instruction Prefetcher	FDIP, 128-entry Fetch Target Queue
L1-I	32KB, 8 ways, 4 cycles latency, LRU, 8 MSHR,
L1-D	48KB, 12 ways, 5 cycles latency, LRU, 16 MSHR
L2	512KB, 8 ways, 12 cycles latency, LRU, 32 MSHR entries
LLC	2MB, 16 ways, 32 cycles latency, LRU 64 MSHR
DRAM	3200MHz, 2 channels, 1 rank, 8 banks, 12.5 (tRP), 12.5 (tRCD), 12.5 (tCAS)

TABLE II: Microarchitectural parameters

Way Count	Target Sizes
1	25
2	5, 25
3	4, 8, 25
4	2, 5, 10, 25
5	0, 4, 7, 12, 25
6	0, 4, 5, 8, 13, 25
7	0, 4, 5, 8, 10, 19, 25
8	0, 4, 5, 7, 9, 11, 19, 25
9	0, 2, 4, 5, 7, 9, 11, 19, 25
10	0, 0, 3, 4, 5, 7, 9, 12, 19, 25
11	0, 0, 3, 4, 5, 7, 8, 10, 16, 21, 25
12	0, 0, 2, 4, 5, 6, 7, 8, 10, 13, 19, 25
13	0, 0, 2, 3, 4, 5, 6, 7, 8, 10, 15, 19, 25
14	0, 0, 0, 3, 4, 5, 6, 7, 8, 9, 11, 16, 20, 25
15	0, 0, 0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 16, 20, 25
16	0, 0, 0, 2, 3, 4, 5, 5, 6, 7, 8, 10, 12, 18, 20, 25
17	0, 0, 0, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10, 13, 18, 21, 25
18	0, 0, 0, 2, 3, 4, 4, 5, 5, 6, 7, 8, 9, 10, 14, 18, 21, 25
19	0, 0, 0, 0, 2, 3, 4, 5, 5, 6, 7, 8, 8, 9, 11, 15, 18, 21, 25
20	0, 0, 0, 0, 2, 3, 4, 4, 5, 5, 7, 7, 8, 9, 10, 11, 16, 19, 21, 25

TABLE III: Branch target field sizes used for BTB-X across different number of ways.

B. BTB storage budgets

To accommodate the rapidly growing branch footprints of contemporary workloads, modern CPUs feature massive BTBs with 10s to 100s of thousands of entries that require hundreds of KB to multiple MB of storage per core. For 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. Similarly, the storage budget in Exynos CPUs nearly doubled with each new generation, growing sixfold over a period of approximately eight years.

We observe that, given the massive BTB sizes, saving a few bits per entry would result in a significant overall storage saving. Furthermore, since tags are now responsible for the bulk of BTB storage requirements, due to the storage-efficient target representations proposed by prior work [4], we study the sizing of the tag field to achieve a balance between storage, aliasing, and the number of BTB entries.

III. Methodology and Terminology

To investigate the impact of BTB tag size on performance and storage efficiency, we use ChampSim [9], [14], a trace driven simulator with detailed implementation of core front-end structures, including BTB and branch direction predictors. In addition, the front-end is equipped with a Fetch Directed Instruction Prefetcher (FDIP) [31]. The microarchitectural parameters of the modeled processor can be found in Table II. For the experiments, we warm up the microarchitectural structures for 50 million instructions and run the simulation for an additional 50 million instructions.

A. Baseline BTB

We use the state-of-the-art BTB design, called BTB-X [4], as the baseline for all studies in the work. BTB-X is a storage-efficient design based on two key insights: 1) Most branches exhibit short offsets; thus, storing offsets instead of full targets reduces storage overhead, and 2) there is a large variability in offset sizes. Therefore, BTB-X stores branch offsets instead of full targets. Further, to accommodate variability in offset size, it employs a set-associative structure in which each way stores offsets of a different size. Therefore, in our studies, we need a mechanism to resize the target/offset fields when varying the number of ways from the default value of eight in the baseline BTB-X. To address this, we employ the same mechanism proposed by BTB-X authors, i.e. the target field sizes are chosen to uniformly distribute the dynamic branches across the ways. Table III lists the target field sizes used for different configurations.

B. Workloads

Our experimental evaluation is based on two sets of traces. The first set consists of traces provided by Qualcomm for the first instruction prefetching championship (IPC-1) [19]. Moreover, we incorporate the modifications proposed by Feliu et al. [12] to address the known issues in these traces. These traces represent workloads from both server and client domains, as well as from SPEC2017 suite. These traces were collected on ARM architectures. The second set of traces comprises 10 workloads obtained from different benchmark suites: *Kafka*, *Tomcat*, and *Spring* from Java DaCapo benchmark suite [5], *Finagle-chirper* and *Finagle-HTTP* from Java Renaissance suite [30], *TPCC*, *Twitter*, and *Wikipedia* from Java BenchBase suite [10], NodeApp (NodeJS online shop webserver), and PHPWiki (PHP wiki web server). These traces were collected on x86 machines.

C. Terminology

Alias or false hit: The BTB configurations in our experiments use partial tags which are smaller than the full tag. One of its implication is that the partial tag can match on a BTB access

Tag Size \ Entries	0.25K	0.5K	1K	2K	4K	8K	16K	32K
Sets	32	64	128	256	512	1024	2048	4096
3b	0.137	0.137	0.139	0.144	0.157	0.194	0.267	0.392
4b	0.185	0.187	0.192	0.197	0.223	0.284	0.402	0.594
5b	0.258	0.267	0.276	0.293	0.334	0.436	0.615	0.867
6b	0.336	0.355	0.383	0.412	0.495	0.657	0.894	1.216
7b	0.405	0.431	0.480	0.545	0.675	0.916	1.237	1.548
8b	0.452	0.485	0.568	0.662	0.844	1.204	1.551	1.797
9b	0.478	0.523	0.624	0.742	0.978	1.440	1.783	1.958
10b	0.496	0.544	0.657	0.792	1.073	1.598	1.926	1.998
11b	0.504	0.554	0.674	0.826	1.133	1.686	1.962	2.104
12b	**0.507**	**0.559**	0.683	0.846	1.159	1.708	2.058	2.104
13b	0.509	0.561	**0.689**	0.854	1.165	1.768	2.058	**2.162**
14b	0.510	0.562	0.691	**0.855**	**1.186**	1.768	**2.103**	2.162
15b	0.510	0.562	0.692	0.863	1.186	**1.787**	2.103	2.162
16b	0.510	0.562	0.693	0.863	1.190	1.787	2.103	2.162
full	0.510	0.563	0.694	0.864	1.190	1.787	2.103	2.162

TABLE IV: Instructions per cycle (IPC) for an 8-way BTB across different number of sets and tag sizes. The bold entries mark performance-oriented design points.

Tag Size \ Entries	0.25K	0.5K	1K	2K	4K	8K	16K	32K
Sets	32	64	128	256	512	1024	2048	4096
3b	785.46	769.69	789.15	769.51	692.42	535.03	362.01	218.07
4b	495.33	493.88	494.09	496.02	435.20	326.73	207.81	120.53
5b	264.01	263.50	272.97	269.98	243.57	178.01	111.76	63.96
6b	137.47	133.83	139.87	146.46	125.94	93.53	58.84	32.74
7b	65.46	68.66	74.00	75.13	66.66	49.37	30.04	16.43
8b	31.44	35.68	36.85	38.45	34.68	25.11	15.10	8.08
9b	15.94	16.17	18.37	20.24	18.70	12.47	7.31	4.27
10b	6.80	7.31	9.08	11.57	9.49	5.82	3.89	3.55
11b	3.00	3.38	5.09	5.78	4.18	3.21	3.21	1.17
12b	1.42	1.65	2.42	2.65	2.44	2.60	0.95	1.17
13b	0.64	0.86	1.06	1.65	2.04	0.54	0.95	0.00
14b	0.32	0.42	0.57	1.41	0.24	0.54	0.00	0.00
15b	0.15	0.20	0.45	0.09	0.24	0.00	0.00	0.00
16b	0.04	0.15	0.02	0.09	0.00	0.00	0.00	0.00

TABLE V: Aliases per kilo instructions (APKI) for an 8-way BTB across different number of sets and tag sizes.

even though the full tag would not have matched. This occurs because two branch PCs may hold the same values in the bits included in the partial tag; while differing in the bits omitted from it. As such, BTB hits arising from partial tag matches, where the corresponding full tags would not have matched, are defined as false hits or aliases.

Hit or true hit: All BTB hits that are not false hits are called hits or true hits. These are the hits resulting from partial tag matches where the full tags would also have matched.

Performance-oriented design: A BTB design with partial tags that achieves more than 99% of the performance of a full tag BTB design.

Storage-efficient design: A BTB design with partial tags that maximizes performance per KB of BTB storage.

IV. Tag Sizing for Performance-oriented BTBs

This section examines the impact of tag size on performance, BTB misses, and aliasing behavior for a performance-oriented BTB design across a range of BTB capacities. The BTB capacity can be scaled by varying either the number of sets or the ways. We first analyze capacity scaling through increasing the number of sets, and then evaluate scaling by adding more ways.

A. Tag sizing with scaling BTB sets

For this study, we vary the BTB capacity from 256 entries to 32K entries by increasing the number of sets from 32 to 4096, in powers of 2, while keeping the number of ways fixed at eight. Further, we vary the tag size from 3 bits to 16 bits. The smallest tag size is three bits because at least three bits are needed to distinguish between the eight ways in BTB.

Tag Size \ Entries	0.25K	0.5K	1K	2K	4K	8K	16K	32K
Sets	32	64	128	256	512	1024	2048	4096
3b	13.10	10.70	8.05	5.25	2.50	1.00	0.44	0.26
4b	37.36	30.96	22.42	14.66	6.67	1.85	0.55	0.27
5b	56.15	46.35	33.91	22.83	10.76	2.64	0.61	0.27
6b	65.67	55.28	40.07	27.29	13.18	3.19	0.65	0.28
7b	71.00	60.08	43.58	29.71	14.77	3.49	0.67	0.28
8b	73.99	62.82	45.13	31.09	15.63	3.65	0.69	0.29
9b	75.34	63.89	45.94	31.80	16.17	3.73	0.69	0.29
10b	75.74	64.54	46.21	32.24	16.44	3.77	0.69	0.29
11b	76.08	64.85	46.31	32.45	16.57	3.78	0.69	0.29
12b	76.19	64.93	46.37	32.57	16.64	3.79	0.70	0.29
13b	76.21	64.96	46.37	32.61	16.66	3.81	0.70	0.29
14b	76.22	64.98	46.41	32.65	16.68	3.81	0.69	0.29
15b	76.24	65.04	46.42	32.66	16.68	3.81	0.69	0.29
16b	76.26	65.03	46.43	32.66	16.70	3.81	0.69	0.29

TABLE VI: Misses per kilo instructions (MPKI) for an 8-way BTB across different number of sets and tag sizes.

The instructions per cycle (IPC), aliases per kilo instructions (APKI), and misses per kilo instructions (MPKI) are presented in Table IV, Table V, and Table VI, respectively. Each row in the tables corresponds to a different tag size, while the number of sets varies across columns, and the metric (IPC, APKI, and MPKI) for the corresponding configuration is presented in the intersecting cells. Also, the top row of Table IV lists the number of BTB entries that correspond to each column of these tables.

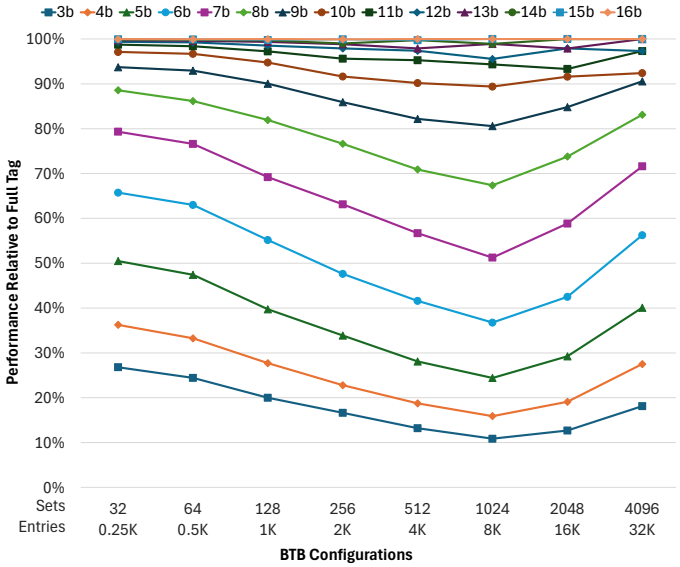

Fig. 2: Performance normalized to full tag performance for an 8-way BTB across different number of sets and tag sizes.

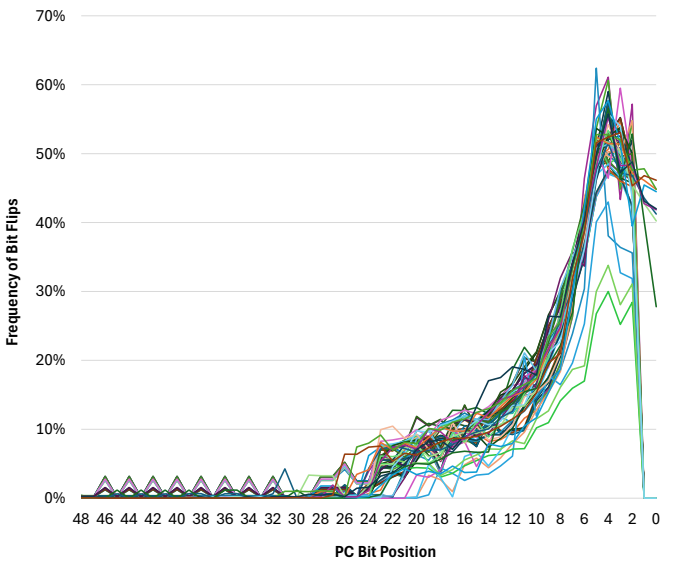

Fig. 3: Frequency of bit flips at different positions in branch PCs in the dynamic instruction stream. Each line is a different workload.

The key finding from this study is:

Finding 1

The tag size required to approach full-tag performance varies non-monotonically with BTB capacity, peaking at moderate BTB sizes.

The results in Table IV indicate that, as we add more sets, the tag size required to achieve at least 99% of the full tag performance increases until we reach 1024 sets and then it starts to decrease. Specifically, 256, 512, 1K, 2K, 4K, 8K, 16K, and 32K-entry BTBs require 12, 12, 13, 14, 14, 15, 14, and 13 bit tags as shown by bold entries in Table IV. Reducing the tag size beyond these leads to rapid increase in aliasing, as shown in Table V, which hurts performance.

Figure 2 visualizes how close we get to the full tag performance for different tag sizes while varying the number of sets. The figure shows that, for any given tag size, the performance of small (64, 128 sets) and large (4096 sets) BTBs is closer to the full tag performance compared to moderately sized BTBs (512, 1024 sets). For example, for a 9-bit tag, 64 and 128 set configurations provide 94% and 93% of full tag performance; whereas 512 and 1024 set configurations achieve only 82% and 80% of full tag performance, and this number increases to 91% for the 4096 set configuration. This result also means that moderate set configurations need larger tags than small and large set configurations to bring the performance within a certain range of full tag performance.

Small BTBs need fewer tag bits because BTB misses are so frequent, as shown in Table VI, that a moderate size tag is sufficient to keep the aliasing low. As the BTB size grows, the number of misses starts to reduce, and the aliasing begins to manifest itself. For example, as Table V reports, a 10-bit tag shows APKI of 6.8 for 256-entry BTB; however, it gradually increases with BTB size and nearly doubles with 2K-entry BTB. Consequently, to offset this increasing APKI, we need to increase the tag size as BTB capacity grows.

Large BTBs require smaller tags because the APKI starts to drop beyond a certain BTB capacity. For example, for a 10-bit tag, the APKI gradually reduces as BTB capacity increases beyond 2K entries. This is because larger BTBs require more bits to index them, which moves the location of tag bits towards the higher-order PC bits. For example, a 256-set BTB needs 8 index bits, so the tag bits start from bit position 9 in the PC. Increasing the number of sets to 4K will require 12 index bits; thus shifting the starting position of the tag bits to bit 13 in PC. For a constant tag size, this implies that for one bit increase in index, one additional PC bit - which earlier was dropped - is now included in the tag. However, as PC bits around bit position 24 and higher do not contain much information, because they switch infrequently (Figure 3), we do not need to include these bits in the tag because doing so does not affect aliasing. Thus, the tag size starts to reduce without increasing aliasing.

Implication 1

The implication of finding above is that increasing the BTB size without adjusting the tag size results either in suboptimal performance or overprovisioned tag storage. Therefore, the tag size should be appropriately adjusted along with changing the number of sets.

B. Tag sizing with scaling BTB ways

For this study, we fix the number of sets in the BTB to 512 and vary the number of ways from 1-way to 20-ways, and the tag size from two bits to 16 bits. We start with two-bit tags because prior work [11] showed that a two-bit tag is sufficient

Tag Size \ Entries	0.5k	1k	1.5k	2k	2.5k	3k	3.5k	4k	4.5k	5k	5.5k	6k	6.5k	7k	7.5k	8k	8.5k	9k	9.5k	10k
Ways	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2b	0.202	0.167	0.146	0.142																
3b	0.296	0.247	0.213	0.193	0.182	0.171	0.161	0.157												
4b	0.396	0.352	0.314	0.285	0.269	0.251	0.234	0.223	0.218	0.216	0.208	0.206	0.203	0.201	0.199	0.198				
5b	0.476	0.459	0.433	0.407	0.388	0.367	0.349	0.334	0.327	0.322	0.312	0.306	0.302	0.297	0.295	0.290	0.286	0.284	0.282	0.279
6b	0.540	0.560	0.555	0.542	0.532	0.518	0.505	0.495	0.485	0.480	0.470	0.464	0.459	0.454	0.449	0.443	0.438	0.433	0.431	0.427
7b	0.577	0.630	0.651	0.662	0.668	0.671	0.673	0.675	0.675	0.676	0.674	0.674	0.672	0.669	0.665	0.660	0.657	0.652	0.652	0.647
8b	0.596	0.671	0.718	0.750	0.777	0.800	0.822	0.844	0.863	0.873	0.887	0.899	0.905	0.910	0.914	0.920	0.921	0.920	0.919	0.916
9b	0.606	0.694	0.754	0.805	0.847	0.888	0.933	0.978	1.020	1.045	1.085	1.114	1.135	1.155	1.170	1.190	1.212	1.217	1.220	1.228
10b	**0.611**	0.704	0.776	0.838	0.892	0.947	1.009	1.073	1.134	1.173	1.235	1.281	1.319	1.353	1.379	1.414	1.454	1.466	1.475	1.493
11b	0.613	**0.711**	0.789	0.858	0.920	0.984	1.057	1.133	1.206	1.255	1.332	1.390	1.439	1.483	1.517	1.564	1.619	1.640	1.653	1.679
12b	0.614	0.714	**0.795**	**0.866**	0.932	0.999	1.077	1.159	1.238	1.292	1.379	1.445	1.502	1.551	1.593	1.646	1.711	1.738	1.756	1.787
13b	0.615	0.715	0.796	0.868	0.934	1.002	1.082	1.165	1.244	1.300	1.389	1.458	1.516	1.567	1.610	1.664	1.734	1.763	1.781	1.815
14b	0.615	0.716	0.801	0.875	**0.943**	**1.015**	**1.099**	**1.186**	**1.270**	**1.329**	**1.424**	**1.496**	**1.557**	**1.614**	**1.661**	**1.722**	1.801	1.833	1.854	1.891
15b	0.615	0.716	0.801	0.875	0.943	1.015	1.099	1.186	1.270	1.329	1.424	1.496	1.557	1.614	1.661	1.722	1.801	1.833	1.854	1.891
16b	0.615	0.716	0.801	0.875	0.944	1.017	1.101	1.190	1.275	1.334	1.431	1.504	1.567	1.626	1.674	1.738	**1.820**	**1.855**	**1.877**	**1.917**
full	0.615	0.716	0.801	0.875	0.944	1.017	1.101	1.190	1.275	1.334	1.431	1.504	1.567	1.626	1.674	1.738	1.821	1.855	1.877	1.917

TABLE VII: Instructions per cycle (IPC) for a 512-set BTB across different number of ways and tag sizes. The bold entries mark performance-oriented design points.

Tag Size \ Entries	0.5k	1k	1.5k	2k	2.5k	3k	3.5k	4k	4.5k	5k	5.5k	6k	6.5k	7k	7.5k	8k	8.5k	9k	9.5k	10k
Ways	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2b	463.1	629.4	759.2	787.7																
3b	231.9	341.6	444.8	516.5	565.2	615.5	668.6	692.4												
4b	114.8	176.8	234.7	286.0	318.9	359.9	403.3	435.2	451.7	458.1	481.1	490.3	498.4	507.8	514.2	518.3				
5b	59.22	93.13	124.5	151.9	173.5	198.9	222.0	243.6	255.3	262.5	278.2	287.8	295.2	303.6	307.9	315.6	323.5	327.6	329.9	335.3
6b	27.58	45.80	62.89	78.73	90.29	103.2	115.3	125.9	135.0	139.6	148.5	154.1	158.5	163.1	167.0	172.0	176.8	180.3	181.9	185.6
7b	13.28	22.24	31.87	40.26	47.02	53.83	60.63	66.66	70.99	73.45	78.13	81.07	83.48	85.84	88.66	87.94	93.52	95.69	95.94	98.52
8b	6.45	10.76	16.17	20.38	24.00	27.77	31.42	34.68	36.81	38.39	41.02	42.52	43.93	45.26	46.28	47.39	49.24	50.21	50.92	52.12
9b	3.09	5.38	8.89	11.06	13.01	15.08	16.96	18.70	19.67	20.45	21.65	22.45	23.13	23.60	24.15	24.68	25.32	25.80	26.13	26.56
10b	1.54	2.77	4.37	5.53	6.61	7.64	8.61	9.49	9.87	10.27	10.84	11.21	11.48	11.70	12.03	12.29	12.67	12.98	13.16	13.36
11b	0.69	1.15	1.95	2.43	2.87	3.46	3.85	4.18	4.38	4.59	4.91	5.06	5.24	5.33	5.54	5.65	5.93	6.10	6.18	6.33
12b	0.32	0.55	1.14	1.43	1.67	2.09	2.28	2.44	2.58	2.66	2.81	2.84	2.89	2.99	3.02	3.13	3.26	3.34	3.35	3.44
13b	0.24	0.42	0.92	1.19	1.39	1.78	1.91	2.04	2.14	2.18	2.30	2.32	2.35	2.43	2.47	2.56	2.66	2.73	2.74	2.80
14b	0.02	0.03	0.04	0.10	0.15	0.17	0.20	0.24	0.30	0.32	0.34	0.39	0.41	0.44	0.47	0.52	0.55	0.60	0.61	0.66
15b	0.02	0.03	0.04	0.10	0.15	0.17	0.20	0.24	0.30	0.32	0.34	0.39	0.41	0.44	0.47	0.52	0.55	0.60	0.61	0.66
16b	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

TABLE VIII: Aliases per kilo instructions (APKI) for a 512-set BTB across different number of ways and tag sizes.

to obtain 99.9% of the performance of a full tag. The IPC and APKI results are presented in Table VII and Table VIII.

The key takeaways from this study are as follows.

Finding 2

Increasing BTB associativity amplifies aliasing and requires larger tags; otherwise, in some cases, higher associativity can degrade performance.

In contrast to the finding of Section IV-A, the bold entries in Table VII show that the tag size required to achieve at least 99% of the full tag performance always increases with the number of added ways. This is because increasing the number of ways does not affect the number of bits required to index the BTB, while increasing the number of sets does. Further, the probability of aliasing increases with the number of ways. For example, a false hit can come from only one location in a direct-mapped BTB; however, it can happen at four locations in a 4-way BTB. Indeed, Table VIII shows a consistent increase in APKI with more ways at all tag sizes.

The impact of aliasing is especially severe at small tag sizes where adding more ways, counter-intuitively, results in performance loss as shown in the region marked by ❶ in Table VII. As the tag size increases beyond 5-bits, aliasing starts to reduce and adding more ways gradually starts to provide performance, region ❷ in Table VII. Even in this

region, IPC drops after a certain way count, e.g. after 10 ways for 7-bit tag.

For tag sizes of 9 or more bits, the aliasing is low enough that the additional ways always result in better performance. However, the higher aliasing resulting from additional ways still limits the performance benefits. For example, for a tag size of 11 bits, increasing the number of ways from two to eight boosts IPC from 0.711 to 1.133. Although the 2-way configuration does not benefit much from increasing the tag size, the 8-way configuration shows a significant performance opportunity from a larger tag. Specifically, for the 2-way configuration, IPC increases from 0.711 to only 0.716, an increase of about 0.7%, even when going to the full tag. In contrast, the 8-way configuration shows an increase of approximately 3%, as the IPC increases from 1.133 to 1.165, by adding just two more tag bits.

Figure 4 visualizes how close we get to the full tag performance for different tag sizes while varying the number of ways. As the figure shows, for a direct mapped BTB, all tag sizes larger than nine bits achieve nearly the same performance as the full tag. However, their performance starts to decline as more ways are added. For instance, an 11-bit tag experiences a 10% drop in normalized performance at the 16-way configuration. This highlights the importance of adjusting the tag size when adding more ways to ensure a performance optimal design.

Implication 2

The implication of this finding is that merely increasing the number of ways, when additional storage is allocated to BTB, would result in suboptimal performance because of the increased aliasing that comes with additional ways. Therefore, the tag size should also be increased appropriately while adding more ways to BTB.

Finding 3

Increasing the number of ways does not always yield better performance than increasing the tag size.

Table VII shows that depending on where a BTB configuration lies in the design space, increasing tag size can provide better performance than adding a way, or vice-versa, even when aliasing is sufficiently low. For example, for an 8-way configuration with 10-bit tags, increasing the tag size by two bits requires about the same storage as adding one additional way. For this configuration, Table VII at ❸ shows that increasing the tag size provides better performance gain (8.0%) than adding a way (5.7%). In contrast, for a 4-way configuration with 10-bit tags, ❹ in Table VII shows that adding a way provides better performance (6.4%) than increasing the tag size by 5 bits (4.4%), with both changes requiring similar additional storage.

The reason for this behavior also lies in aliasing. As Table VIII shows, the 4-way configuration shows about half the aliasing of the 8-way configuration. Therefore, increasing tag size helps more in the 8-way configuration than in the 4-way one. Consequently, the 8-way configuration favors increasing tag size while the 4-way configuration favors the additional way.

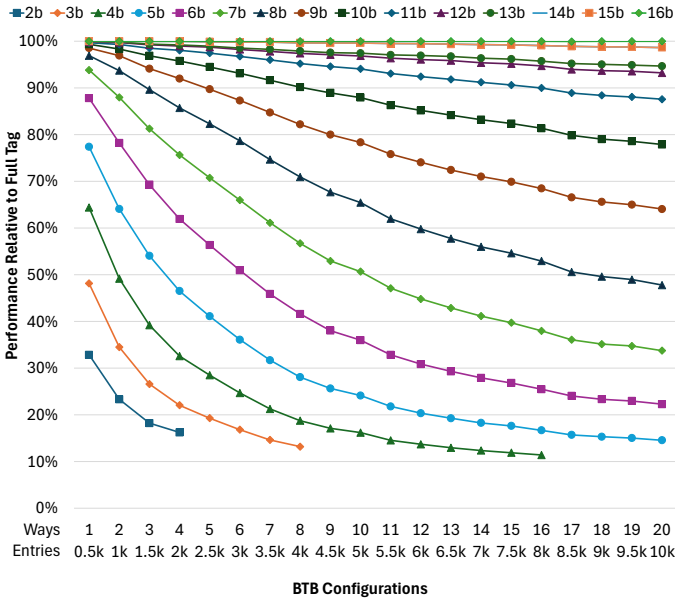

Fig. 4: Performance normalized to full tag performance for a 512-set BTB across different number of ways and tag sizes.

Implication 3

These results imply that, if additional storage budget is allocated to BTB, careful analysis is required to determine whether to invest it in enlarging the tags or adding more ways (or both).

V. Tag Sizing for Storage-Efficient BTBs

This section explores the impact of tag size on storage efficiency and performance of a storage-efficient BTB design across a range of BTB capacities. We vary the BTB capacity by varying the number of ways and sets.

Finding 4

Storage-efficient BTB designs require two to four fewer tag bits compared to performance-oriented designs. As a result, they reduce tag storage requirements by up to 21% and overall BTB storage by up to 14%; however, they achieve only 90%-97% of the full tag performance, as opposed to the 99% by the performance-oriented designs.

Tag sizing with scaling BTB sets: Figure 5 presents the storage efficiency, i.e. IPC/KB of BTB storage, as the number of sets is varied from 32 to 4096 for an 8-way BTB, across a range of tag sizes. The ♦ in the figure shows that the most storage-efficient design points require 9, 10, 10, 11, 11, 11, 10, and 9 bit tags as the number of sets increases from 32 to

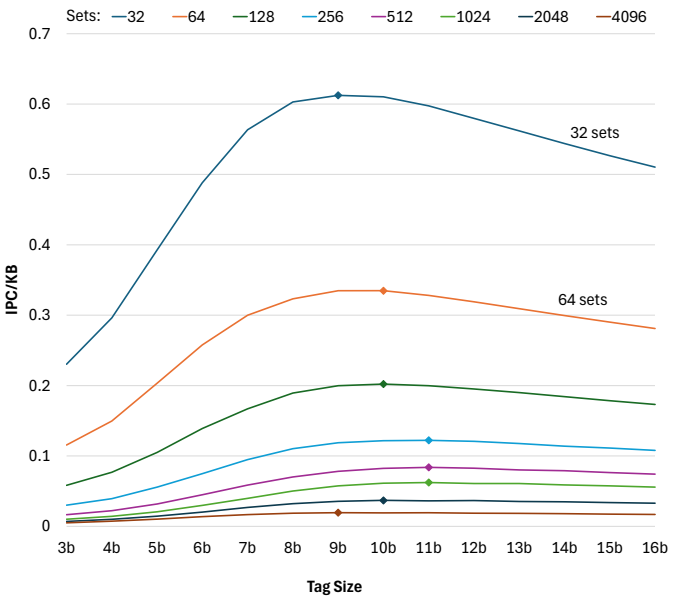

Fig. 5: Storage efficiency for an 8-way BTB across different number of sets and tag sizes. The ♦ indicates the most storage-efficient configurations.

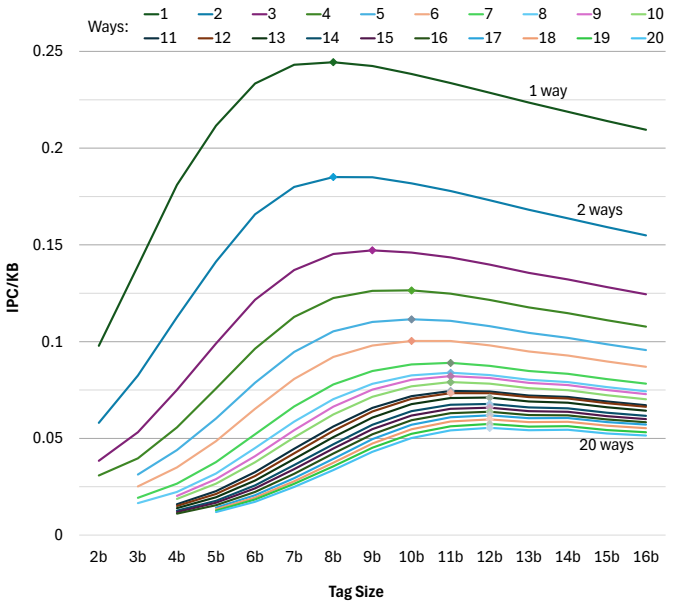

Fig. 6: Storage efficiency for a 512-set BTB across different number of ways and tag sizes. The ♦ indicates the most storage efficient configurations.

4096 in powers of 2, respectively. In contrast, the performance-oriented BTBs require 12, 12, 13, 14, 14, 15, 14, and 13 bit tags for the same number of sets. Therefore, depending on BTB capacity, the storage-efficient designs save two to four tag bits, which reduces tag storage by 11%-21% and overall BTB storage by 7%-14% compared to the performance-oriented designs. However, they achieve only 90%-97% of the full tag performance, due to more aliasing, compared to more than 99% achieved by the performance-oriented designs.

Figure 5 also shows that the storage efficiency improves with larger tags; however, beyond a certain tag size, storage-efficiency starts to decline. This is because the increase in tag size begins to yield diminishing returns in performance once the aliasing is sufficiently low. For any further increase in tag size, the increase in storage requirements outweighs the performance improvement, thus leading to a decrease in storage efficiency. Further, the figure also shows that the tag size requirements for storage-efficient designs show the same trends as for the performance-oriented ones in Table IV, as the number of sets is scaled. Specifically, the tag size required to achieve the most storage-efficient design, marked with ♦, first increases until the number of sets reaches 256-1024, and the it starts to reduce.

Tag sizing with scaling BTB ways: Figure 6 illustrates how storage efficiency changes as the number of ways increases from 1 to 20 for a 512-set BTB across different tag sizes. Similar to set scaling, the tag sizes for the storage-efficient design are two to three bits smaller than those for a performance-oriented one with way scaling as well. Specifically, storage-efficient designs require 8, 8, 10, 11, and 12 bit tags for direct mapped, 2-way, 4-way, 8-way, and 16-way BTBs compared to the 10, 11, 12, 14, and 14 bits required by the performance oriented design for the same associativity. By saving two to three tag bits, they reduce tag storage requirements by 10%-20% and overall BTB storage requirements by 5% to 14%. However, this storage-efficiency comes at a performance penalty of 2% to 7%.

Like set scaling, the storage efficiency improves with larger tags while scaling the number of ways as shown in Figure 6, and it starts to drop beyond a particular tag size. Further, the figure also shows that the tag size requirements for storage-efficient designs show the same trends as for the performance-oriented ones in Table VII, as the number of ways is scaled. Specifically, the tag size required to achieve the most storage-efficient design, marked with ♦, increases with the number of ways due to the increased aliasing that the additional ways bring with them.

VI. Are the findings generalizable?

This section assesses whether the findings of Section IV and Section V hold across different core front-end configurations with different BTB designs, branch predictors, and tag folding mechanisms. Specifically, we evaluate five additional front-end configurations:

Two-level BTB: A 2-level BTB hierarchy with a 128-entry L1 BTB (single-cycle latency) and the L2 BTB with the following capacities and latencies: 256-entry BTB with 2 cycle latency, 512-, 1K-, and 2K-entry BTBs with 3 cycle latency, 4K-entry BTB with 4 cycle latency, 8K-entry BTB with 5 cycle latency, and 16K- and 32K-entry BTBs with 6 cycle latency.

GShare: It use a gShare branch predictor instead of TAGE.

Configuration		256-entry	512-entry	1K-entry	2K-entry	4K-entry	8K-entry	16K-entry	32K-entry
2-level BTB	Perf-oriented	12	12	13	13	14	13	12	12
	Storage-efficient	9	9	10	10	10	11	10	9
GShare	Perf-oriented	11	12	12	13	14	13	12	11
	Storage-efficient	9	9	9	10	11	10	9	9
Folded Tags	Perf-oriented	12	12	12	13	13	14	13	12
	Storage-efficient	9	9	10	10	11	11	10	10
PDede	Perf-oriented	12	12	13	13	14	15	14	13
	Storage-efficient	10	10	10	11	11	11	10	9
Conv-BTB	Perf-oriented	11	12	12	13	14	14	14	13
	Storage-efficient	9	9	9	10	11	11	10	9

TABLE IX: Tag size requirements across different front-end configurations and BTB capacities for performance-oriented and storage-efficient BTB designs.

Folded tags: Instead of truncating the higher order tag bits, this configuration folds them, as introduced in [3]. Specifically, it retains the least significant 8 bits of the full tag, which have the highest entropy, and uniformly folds the remaining bits to achieve the desired tag size. For example, for a 12-bit folded tag, the lower 8 bits are kept intact, while the remaining bits are XORed in groups of 4-bits to get the upper 4 bits.

PDede: This configuration use PDede [36] as the baseline BTB instead of BTB-X.

Conv-BTB: This configuration uses a conventional BTB that stores full targets as the baseline instead of BTB-X.

For each configuration, Table IX reports the tag size required for both performance-oriented (99% of full tag IPC) and storage-efficient (maximum IPC/KB) designs across BTB capacities ranging from 256 to 32K entries. The table shows that our findings hold for all the additional configurations. These configurations differ slightly only in the absolute number of tag bits they require. Specifically, the table shows that Finding 1 holds as, for every configuration, the tag size requirement peaks at moderate BTB capacities (typically at 4K- or 8K-entry BTB) and then decreases for larger BTBs, while small BTBs also require fewer tag bits than moderate-sized ones. For example, the tag size for PDede peaks at 15 bits for the 8K-entry configuration and reduces to 13 bits at 32K entries, closely mirroring the behavior observed for BTB-X.

The data in Table IX further corroborates Finding 4, namely that storage-efficient designs consistently require fewer tag bits than performance-oriented designs. Across all five configurations and all BTB capacities, the storage-efficient design requires two to four fewer tag bits compared to the performance-oriented design. For instance, for GShare at the 8K-entry point, the performance-oriented design requires 13 bits while the storage-efficient design needs only 10 bits, a reduction of three bits. This gap is consistent across configurations, reinforcing the finding that storage-efficient BTB designs can achieve meaningful reductions in tag storage overhead relative to performance-oriented ones.

In summary, these results show that our finding generalize across a wide range of modern front-end configurations..

VII. Future Research Directions

Following are some of the research directions that we believe this work will stimulate:

Research Direction 1

BTB designs for minimizing tag storage.

This work draws research community's attention to an emerging bottleneck in BTB design, i.e. the growing storage overhead of tags, which now dominates overall BTB storage requirements. We expect these findings to motivate future research into novel tag representations that reduce tag storage overhead.

Research Direction 2

Analytical models for tag sizing.

The key insight from this work is that the tag size requirements vary with the number of BTB ways, sets, and if the designers opt for performance oriented or storage efficient design. Consequently, designers would need to run a large number of simulations to determine the optimal tag size every time they change BTB configuration. To cut down this redundant work, we expect the community to investigate analytical models to reason about tag sizing. Depending on accuracy, a model will either eliminate the need for simulation sweeps or reduce the search space, thus reducing the number of simulations. For example, if the model says that the tag size can only reduce after a certain point, there won't be any need to simulate larger tag sizes. Further, the model can be used to determine, if additional storage becomes available, whether it should be invested in increasing tag size, the number of entries, or distributed between the two.

Research Direction 3

Customizing BTB indexing functions.

In Section IV-A, we make a key observation that, while scaling BTB sets, the tag size starts to reduce as the number of index bits (and BTB sets) increase beyond a certain value. Further, Figure 3 shows that different PC bits exhibit different

entropy. We expect these results to ignite research in BTB indexing functions that, instead of using lower order PC bits, use high entropy PC bits, possibly with a dynamic selection mechanism, for indexing. This will reduce the tag size requirements because the remaining bits would have a low entropy.

VIII. Related Work

Core front-end is a major performance bottleneck for many contemporary applications. To address this bottleneck, researchers have optimized many front-end components including BTB, instruction cache(L1-I), branch direction predictor, etc. BTB is one of the most important front-end components and lies on the critical path for instruction delivery to the core and instruction prefetching. Prior work has explored several aspects of BTB design, namely its organization, prefetching, replacement policies etc.

From the BTB organization perspective, prior work [3], [4], [18], [33], [34], [36] have explored a number of ways to reduce branch target storage requirements. Seznec et. al. [33], [34] observed that all the branch targets in the same page shared the page number. Therefore, they proposed to store the target page number in a separate structure, and the BTB stores only a pointer to this structure in addition to the offset of branch target within the page. This organization reduces storage cost as the pointer requires fewer bits in BTB compared to the page number. Another design, called PDede [36], observed that if the branch target is within the same page as the branch instruction itself, the target page number need not be stored as it can be extracted from branch PC. PDede reserves half of the BTB entries for the same page branches, and these entries do not offer storage for storing pointers to page numbers, thus reducing BTB storage requirements. Micro BTB [17] uses flexible entries that adaptively store either two small-offset targets or one large-offset target. A recent design, called BTB-X [4], leverages the offset distribution across workloads, resulting in different fixed-sized ways, optimized for common offset ranges. This achieves a dense packing of the target offsets, thus reducing storage requirements.

Regarding prior work on reducing BTB tag size requirements, early work by Fagin [11] demonstrated that BTBs do not need to store full tags. They found that on average 2-bit tags provide 99.9% of full tag performance and even in the worst case a 4-bit tag leaves you with 99.9% performance.

Apart from optimizing BTB organization, prior work [6], [8], [22], [24], [25] has also explored BTB prefilling/prefetching to mitigate BTB misses. Twig [22] is a profile guided software prefetcher that analyzes an application's execution profile to identify critical BTB misses and then injects software prefetch instructions. The prefetch instruction takes compressed branch PC and target as operands and its execution fills this information in BTB. PhantomBTB [8] virtualizes branch information in LLC and prefetches it to L1 BTB. Boomerang [25] aims to prefill BTB from cache hierarchy on discovering BTB misses. Shotgun [24] prefills a specialized conditional branch BTB by predecoding L1-I blocks as they are prefetched. Prior work has also explored BTB replacement policies [27], [35]. Furthermore, AVM-BTB [26] shares storage space between BTB and instruction cache.

Instruction prefetching has also been explored to address the front-end bottleneck in general. Temporal stream prefetchers [13], [20], [21] have been very effective in eliminating instruction cache misses; however, their high storage overhead makes them unattractive. Therefore, recent research focuses on fetch directed instruction prefetching [31] and its variants [15], [23]–[25], [28] to improve L1-I performance with minimal storage cost. Finally, recent work [7] has also explored storage-efficient L1-I organizations to minimize the unused bits in L1-I cache blocks.

IX. Conclusion

BTB is a critical front-end component for driving both instruction fetch and instruction prefetch. However, the BTB capacity needs to be sufficiently large to achieve a high performance front-end. However, the continuously increasing branch working sets of contemporary application put immense pressure on front-end components including BTBs. Consequently, commercial processors feature huge BTBs with storage requirements reaching into multiple megabytes. Much of this storage budget have been used from storing branch targets. However, recent work in optimizing branch target representation in BTB has substantially reduced the storage requirements of branch targets. Indeed, branch targets no longer dominate BTB storage requirements. It is now the tag size that has become the most storage intensive field in the BTB.

This is the first work that comprehensively studies the impact of tag size on performance, aliasing, and storage requirements of modern BTB organizations. Our results demonstrate that BTB performance optimization requires a careful approach to tag sizing. Depending on the storage constraints, the design objectives and the BTB's organization, different tag sizes are necessary to achieve high performance at a given storage budget. The common practice in prior work of using fixed 12-bit tags across all BTB configurations and organizations might lead to suboptimal performance, as no single tag size can effectively serve the diverse requirements of different BTB designs.

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] "IBM z16 (3931)technical guide. Section 3.4.3, page 84," https://www.redbooks.ibm.com/redbooks/pdfs/sg248951.pdf.

[2] 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, 2020.

[3] 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.

[4] 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.

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[6] J. Bonanno, A. Collura, D. Lipetz, U. Mayer, B. Prasky, and A. Saporito, "Two Level Bulk Preload Branch Prediction," in *International Symposium on High-Performance Computer Architecture*, 2013, pp. 71–82.

[7] R. Brunner and R. Kumar, "Weeding out front-end stalls with uneven block size instruction cache," in *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '24. IEEE Press, 2024, p. 1382–1396. [Online]. Available: https://doi.org/10.1109/MICRO61859.2024.00102

[8] 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

[9] "ChampSim Simulator," https://github.com/ChampSim/ChampSim.

[10] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: an extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, no. 4, p. 277–288, Dec. 2013. [Online]. Available: https://doi.org/10.14778/2732240.2732246

[11] 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.

[12] J. Feliu, A. Perais, D. A. Jiménez, and A. Ros, "Rebasing Microarchitectural Research with Industry Traces." Institute of Electrical and Electronics Engineers (IEEE), 10 2023, pp. 100–114.

[13] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *International Symposium on Microarchitecture*, 2011.

[14] 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.

[15] 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

[16] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, 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.

[17] 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

[18] J. Hoogerbrugge, "Cost-efficient branch target buffers," in *Euro-Par 2000 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 950–959.

[19] "1st Instruction Prefetching Championship Traces," https://research.ece.ncsu.edu/ipc/infrastructure/#Traces.

[20] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared History Instruction Fetch for Lean-core Server Processors," in *International Symposium on Microarchitecture*, 2013.

[21] 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.

[22] 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

[23] 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

[24] 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

[25] 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.

[26] 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 - International Symposium on Computer Architecture*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 17–31.

[27] 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.

[28] 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.

[29] 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.

[30] A. Prokopec, A. Rosà, D. Leopoldseder, G. Duboscq, P. Tůma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder, "Renaissance: Benchmarking suite for parallel applications on the jvm," in *Programming Language Design and Implementation*, 2019.

[31] 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.

[32] A. Saporito, "The IBM z15 processor chip set," in *Hot Chips*, 2020.

[33] 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

[34] A. Seznec, "A 64-kbytes ittage indirect branch predictor," *J. Instruction-Level Parallelism*, 2011.

[35] 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.

[36] 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