BTB-X: A Storage-Effective BTB Organization

Truls Asheim, Boris Grot, and Rakesh Kumar

Abstract—Many contemporary applications feature multi-megabyte instruction footprints that overwhelm the capacity of branch target buffers (BTB) and instruction caches (L1-I), causing frequent front-end stalls that inevitably hurt performance. BTB is crucial for performance as it enables the front-end to accurately resolve the upcoming execution path and steer instruction fetch appropriately. Moreover, it also enables highly effective fetch-directed instruction prefetching that can eliminate many L1-I misses. For these reasons, commercial processors allocate vast amounts of storage capacity to BTBs. This letter aims to reduce BTB storage requirements by optimizing the organization of BTB entries. Our key insight is that today's BTBs store the full target address for each branch, yet the vast majority of dynamic branches have short offsets requiring just a handful of bits to encode. Based on this insight, we organize the BTB as an ensemble of smaller BTBs, each storing offsets within a particular range. Doing so enables a dramatic reduction in storage for target addresses. We also compress tags to reduce the tag storage cost. Our final design, called BTB-X, uses an ensemble of five BTBs with compressed tags that enables it to track 2.8x more branches than a conventional BTB with the same storage budget.

Index Terms—Server, microarchitecture, branch target buffer (BTB), instruction cache, prefeteching

1 Introduction

Contemporary server applications feature massive instruction footprints stemming from deeply layered software stacks. These footprints may far exceed the capacity of the branch target buffer (BTB) and instruction cache (L1-I), resulting in the so-called front-end bottleneck. BTB misses may lead to wrong-path execution, triggering a pipeline flush when misspeculation is detected. Such pipeline flushes not only throw away tens of cycles of work but also expose the fill latency of the pipeline. Similarly, L1-I misses cause the core front-end to stall for tens of cycles while the miss is being served from lower-level caches.

BTB stands at the center of a high-performance core front end for three key reasons: it determines the instruction stream being fetched, it identifies branchs for the branch predictor, and it affects the L1-I hit rate. Specifically, by identifying control flow divergences, the BTB ensures that the branch predictor can make predictions for upcoming conditional branches. For predicted-taken and unconditional branches, the BTB supplies targets to which instruction fetch should be redirected. Finally, the BTB together with the direction predictor enables an important class of instruction prefetchers called fetch-directed instruction prefetchers (FDIP) [6], [7], [9], which rely on the BTB to discover L1-I prefetch candidates.

Considering the criticality of capturing the large branch working sets of modern workloads, commercial CPUs feature BTBs with colossal capacities, a trend also observed by [5]. Thus, IBM z-series processors [3], AMD Zen-2 [11], and ARM Neoverse N1 [8] feature 24K-entry, 8.5K-entry, and 6K-entry BTBs. With each BTB entry requiring 10 bytes or more (Section 2), BTB storage costs can easily reach into tens and even hundreds of KBs. Indeed, the Samsung Exynos M6 mobile processor allocates a staggering 529KB of on-chip storage to BTBs [4]. While such massive BTBs are effective at capturing branch working sets, they do so at staggering area costs.

This work seeks to reduce BTB storage requirements by increasing its *branch density*, defined as branches per KB of storage. To that end, we aim to reorganize individual BTB entries to minimize their storage cost. Our key insight is that branch offsets, defined as delta between the address of the branch instruction and that of its target, are unequally distributed but tend to require significantly fewer bits to represent than full target addresses. Our analysis reveals that 37% of dynamic branches require only 7 bits or fewer for offset encoding, while a meager 1% of branches need 25 bits or more to store their offsets.

Based on this insight, we propose to store offsets in the BTB rather than full target addresses, which can be up to 64 bits long depending on the size of virtual address space. To accommodate the varied distribution of branch offsets, we partition the BTB into several smaller BTBs, each storing only those branches whose target offsets can be encoded with a certain number of bits. Because the target field accounts for over half of each entry's storage budget in a conventional BTB (Fig. 1), this optimization brings significant storage savings. We further observe that the tag field is the second-largest contributor to each BTB entry's storage requirement. To reduce this cost, we propose compressing the tags through the use of hashing.

Our final design, called *BTB-X*, uses an ensemble of five BTBs, each with 16-bit tags. The BTBs differ only in the number of bits they allocate for branch target offsets. Our evaluation shows that BTB-X can track over 2.8x more branches than a conventional BTB with the same storage budget. Conversely, BTB-X can accommodate the same number of branches as existing BTBs while requiring 2.8x less storage.

2 Background

2.1 Branch Target Buffer (BTB)

BTB is used in the core front-end to identify whether a program counter (PC) corresponds to a branch instruction before the instruction itself is even fetched. As depicted in Fig. 1, each BTB entry is composed of *tag*, *type*, and *target* fields. BTB is indexed with the lower order PC bits and *tag* field of the indexed entry is compared with the higher order PC bits. A match indicates that the PC belongs to a branch instruction. The *type* field of the indexed BTB entry determines whether the branch is a call, return, conditional, or unconditional branch. The branch type determines whether the branch direction (*taken/not taken*) needs to be predicted and where its target address is found. Call, return, and unconditional branches are always *taken*, whereas for conditional branches, a direction predictor is used to predict their direction. If the branch is predicted to be taken, *target* field in the BTB entry provides the address for the next instruction, except for returns. This is because a given function can be called from different call sites; as such, the return address is call-site dependent. Therefore, a return address stack (RAS) is typically employed to record return addresses at call-sites. On a function call, the call instruction pushes the return address to RAS, which is later popped by the corresponding return instruction.

2.2 The Cost of a BTB Miss

A BTB miss for a branch instruction means that the branch is undetected and the front-end continues to fetch instructions sequentially. Whether or not the sequential path is the correct one depends on the actual direction of the missed branch. Unless the missed branch is a conditional branch that is not taken, the

- *Truls Asheim and Rakesh Kumar are with the Norwegian University of Science and Technology, 7491 Trondheim, Norway. E-mail: {truls.asheim, rakesh.kumar}@ntnu.no.*
- *Boris Grot is with the University of Edinburgh, Edinburgh EH8 9YL, U.K. E-mail: boris.grot@ed.ac.uk.*

Manuscript received 30 Mar. 2021; revised 3 June 2021; accepted 21 June 2021. Date of publication 3 Sept. 2021; date of current version 4 Oct. 2021.
(Corresponding author: Rakesh Kumar.)
Digital Object Identifier no. 10.1109/LCA.2021.3109945

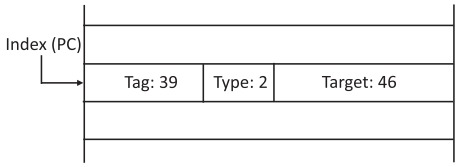


Fig. 1. BTB entry composition in a conventional BTB.

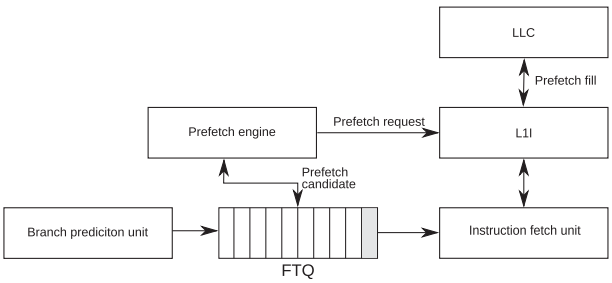


Fig. 2. FDIP microarchitecture.

sequential path is incorrect. When the wrong path is eventually detected by the core, all the instructions after the branch that missed in the BTB are flushed, fetch is redirected to the branch target and pipeline is filled with correct-path instructions. BTB misses are thus highly deleterious to performance as they result in a loss of tens of cycles of work and expose the pipeline fill latency.

2.3 BTB's Role in Instruction Prefetching

Fetch-directed instruction prefetchers are a class of powerful L1-I prefetchers that intrinsically rely on a BTB. These prefetchers are highly effective and, when coupled with a sufficiently large BTB, outperform the winner of the recently-concluded Instruction Prefetching Championship [2], as reported by *Ishii et al.* [5]. Variants of these prefetchers have been adopted in commercial products, for example in IBM z15 [10], ARM Neoverse N1 [8] etc.

Fig. 2 shows a canonical organization of a fetch-directed instruction prefetcher (FDIP) [9]. As originally proposed, FDIP decouples the branch-prediction unit and the fetch engine via the *fetch target queue (FTQ)*. This decoupling allows the branch prediction unit to run ahead of the fetch engine and discover prefetch candidates by predicting the control flow far into the future. With FDIP, each cycle, the branch prediction unit identifies and predicts branches to anticipate upcoming execution path and inserts corresponding instruction addresses into the FTQ. Consequently, the FTQ contains a stream of anticipated instruction addresses to be fetched by the core. The prefetch engine scans the FTQ to identify prefetch candidates and issue prefetch requests.

For FDIP to be effective, the BTB needs to accommodate the branch working set, otherwise frequent BTB misses will cause FDIP to prefetch the wrong path as FTQ will be filled with wrong path instruction addresses. This is one of the key reasons why commercial processors deploy massive BTBs, as also observed by [5].

3 BTB-X

To reduce the overall storage cost, this work seeks to minimize the storage requirements of the costliest fields making up each BTB entry, i.e. target and tag, through two ideas: partitioning and hashing.

3.1 Partitioned BTB

As Fig. 1 shows, the largest contributor to storage cost is the target field, which stores the branch target address. For instance, in the ARMv8 ISA, which uses a 32-bit fixed length instruction encoding, the target address is 46 bits long with a 48-bit virtual address space. Our key insight is that targets of most branches lie relatively close

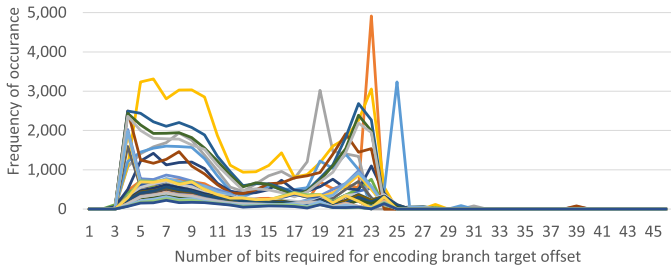


Fig. 3. Distribution of branch target offsets.

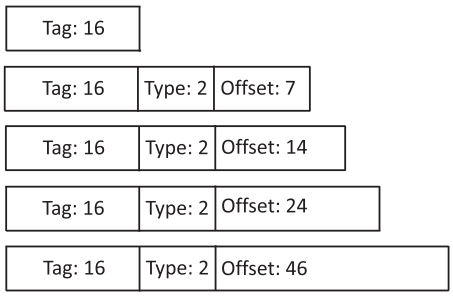


Fig. 4. BTB entry composition for BTB-X partitions.

in the virtual address space to the branch itself. As a result, encoding the *distance* to the target, in the form of an offset from the branch instruction, instead of a full target address, can provides drastic storage savings.

Fig. 3 plots the distribution of offsets in the branch working sets of our workload traces. Offsets are calculated in instruction words, which are 32 bits in the ARM v8 ISA. The data includes both conditional and unconditional branches; hence, it comprehensively covers the full branch working set. The X-axis shows the number of bits required to encode the offset, while the Y-axis plots the frequency of occurrence. Note that, in addition to bits for encoding the offset, an additional bit is required for the direction of the offset (forward/backward).

As the figure shows, short offsets dominate the distribution with 37% of branches requiring only seven bits or fewer for their offsets. A further 30% of branches only require between 8 and 14-bits to represent their offsets. The reason why such a high fraction of offsets is short is that conditional branches dominate the dynamic branch working set, and they tend to have short offsets [6]. This is because conditional branches generally guide the control flow only inside a function; meanwhile, software engineering principles favor small functions, thus restricting conditional branch offsets to short distances.

Perhaps surprisingly, Fig. 3 also shows that very few branches require a large number of bits to encode their offset. Indeed, a meagre 1% of branches requires 25 bits or more for their offset encoding. The sum of these results indicates that reserving space for the full 46-bit target address results in an appalling under-utilization of BTB storage, since 99% of branches need at most half the number of bits needed to represent the full target address if offsets are used instead.

Based on these insights, we propose to partition a single logical BTB into multiple physically-separate BTBs. The BTBs differ amongst themselves only in the size of the offset. When the branch prediction unit queries an address, all BTB partitions are accessed in parallel, hence presenting a logical equivalent of a monolithic BTB. If the core queries the BTB with n addresses per cycles, each BTB-X partition must be accessed with all n addresses.

Fig. 4 shows the BTB partitions used by our proposed BTB organization, called BTB-X. It uses five different BTBs with offset field sizes of 0, 7, 14, 24 and 46 bits. The BTB with no offset field (i.e., 0-bit offset) tracks only return instructions. Recall from Section 2 that return instructions read their target address from RAS; as such,

TABLE 1
Microarchitectural Parameters

Core	6-wide OoO, 128-entry FTQ, 128 reservation stations, 352-entry ROB, 128-entry load queue, 72-entry store queue
Branch Predictor	Hashed Perceptron
L1-I	32 KB, 8-way, 4 cycle latency, 8 MSHRs
L1-D	48 KB, 12-way, 5 cycle latency, 16 MSHRs
L2	512 KB, 8-way, 14/15 cycle latency, 32 MSHRs
LLC	2MB, 16-way, 34/35 cycle latency, 64 MSHRs

TABLE 2
Storage Breakdown for Conventional BTB

Entries	Organization	Entry size (bits)	Total (bytes)
1K	128-set, 8-way	87	10.875K
2K	256-set, 8-way	86	21.5K
4K	512-set, 8-way	85	42.5K
8K	1024-set, 8-way	84	84K
16K	2048-set, 8-way	83	166K

there is no need to allocate space for targets of returns in the BTB. Further, as all instructions in this BTB are returns, it does not require the branch *type* field either. Other branches are allocated entries in one of the remaining four BTBs based on the minimum number of bits required to encode their offsets. For example, if a branch requires 10 bits for encoding its target offset, it is allocated an entry in the BTB with target offset field size of 14 bits.

We further make use of the data in Fig. 3 to size each of the BTBs. Because very few branches require more than 24 bits to encode their target offsets, the BTB with the 46-bit offset field is allocated the fewest entries. Meanwhile, the BTBs corresponding to 7-, 14-, and 24-bit offset are allocated a similar number of entries, as the frequency of 1-7 bit, 8-14 bit, and 15-24 bit offsets is about same – 37%, 30% and 32% respectively.

3.2 Tag Compression

Tags comprise the second largest source of storage overhead in each BTB entry, requiring 39 bits in the baseline design. To further reduce the storage requirement, BTB-X uses a compressed 16-bit tag in all of its BTBs. Our compression scheme maintains the 8 low-order bits same as in the full tag. The remaining bits of the full tag are folded, using the XOR operator, in blocks of eight to compute the 8 higher-order bits for the compressed tag. As our evaluation shows, the performance impact of this scheme is negligible as the hashing function (folded XOR) preserves most of the entropy found in the high-order bits.

3.3 Applicability to Basic-Block-Based BTBs

While this work describes BTB-X in the context of an instruction-based BTB organization (i.e., the BTB is accessed using individual instruction addresses), our insights and design are equally applicable to basic-block-based BTBs (BB-BTBs) [6], [7], [9]. BB-BTBs are similar to instruction-based ones but are accessed using a basic-block address. Because existing BB-BTB designs store full branch targets and offsets, they would benefit from optimizations described in this work.

4 Evaluation

We use ChampSim [1], an open-source trace-driven simulator, to evaluate the efficacy of BTB-X on server and client workload traces from IPC-1 [2]. We warm up microarchitectural structures for 50M instructions and collect statistics over the next 50M. The microarchitectural parameters for the modeled processor are listed in Table 1.

TABLE 3
Storage Breakdown for BTB-X. The Storage Budget is Comparable to That of a 1K-Entry Conventional BTB

Partition	Entry size	Entries	Storage
0-bit offset	16-bits	768	1.5KB
7-bit offset	25-bits	768	2.34KB
14-bit offset	32-bits	640	2.5KB
24-bit offset	42-bits	640	3.28KB
46-bit offset	64-bits	80	0.625KB
Total		2,896	10.25KB

TABLE 4
Storage and Entries in Conventional BTB and BTB-X

Conventional BTB		BTB-X	
Storage	Entries	Storage	Entries
10.875KB	1K	10.25KB	2,896
21.5KB	2K	20.5KB	5,792
42.5KB	4K	41KB	11,584
84KB	8K	82KB	23,168
166KB	16K	164KB	46,336

4.1 Storage Breakdown

The storage requirements for a conventional BTB for different number of BTB entries are presented in Table 2 assuming a 48-bit virtual address space. We increase the number of sets in the BTB to increase the number of entries while keeping the associativity same (8-way). Notice that the entry size reduces by one bit while doubling the number of entries. This is because the tag size reduces as more bits are needed to index the BTB.

Table 3 presents the allocation of the storage budget among the five BTB-X partitions. For this analysis, the storage budget is capped at that of a 1K-entry conventional BTB. As the table shows, the partition for 46-bit offsets gets the smallest amount of storage as very few branches need to be allocated there. Meanwhile, the remaining partitions get relatively more storage with a roughly similar number of entries in each partition.

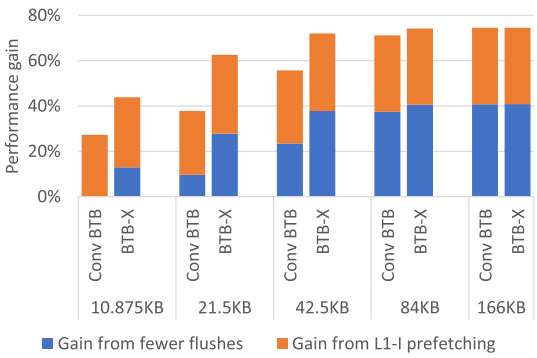
When presented with a larger storage budget, we follow the same strategy for scaling up BTB-X as for a conventional BTB. Thus, we double the number of sets in each BTB partition to double the capacity while maintaining the associativity (i.e., 0-bit and 7-bit offset partitions are 6-way, others are 5-way).

Table 4 shows the number of entries that a conventional BTB and BTB-X can accommodate for several storage budgets. As is evident from the table, for a given storage budget, BTB-X can store about 2.8x more entries than the conventional BTB. Note that since the number of sets have to be a power of 2, we are not able to precisely match the storage of conventional BTB and BTB-X – the conventional BTB gets a slightly higher storage.

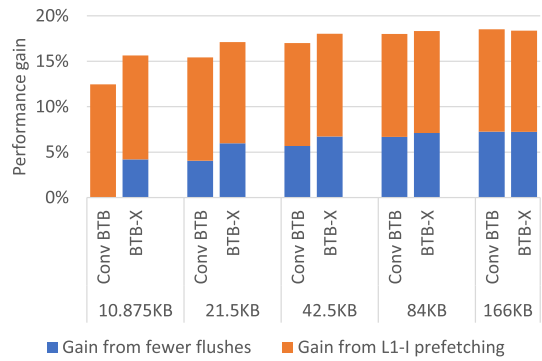
4.2 Performance

To assess the effectiveness of BTB-X, we compare its performance to that of a conventional BTB across different storage budgets. Recall from Section 2 that a larger BTB can deliver two distinct benefits: 1) reduce the incidence of pipeline flushes by detecting branches in the upcoming control flow and 2) facilitate instruction prefetching when coupled with FDIP. Thus, we compare the performance gains achieved by the two competing BTB designs by evaluating them with FDIP.

Fig. 5 presents the performance gains obtained on server and client traces. Each bar in the figure shows the contribution to performance of having fewer pipeline flushes and from better instruction prefetching stemming from larger BTB capacities. The results



(a) Server workloads.



(b) Client workloads.

Fig. 5. Performance gain for conventional BTB and BTB-X (both with FDIP) on (a) *server* and (b) *client* traces. Baseline is no-prefetch 1K-entry conventional BTB. X-axis is storage for a 1K-, 2K-, 4K-, 8K-, and 16K-entry conventional BTB.

are normalized to the performance of a core with a 1K-entry conventional BTB (10.875KB storage budget) and no instruction prefetching.

As the figure shows, BTB-X provides significantly higher overall performance than the conventional BTB for equal storage budgets of up to several tens of kilobytes. The performance advantage of BTB-X is particularly pronounced on server traces whose large instruction footprints pressure the BTB and L1-I. For instance, BTB-X provides 63% performance gain over the baseline compared to 38% of conventional BTB with 21.5KB storage budget. At large BTB storage budgets, the branch working sets of many workloads start to fit in the available BTB capacity, at which point the performance gap between the two designs diminishes.

A key take-away from the figure is that BTB-X provides same or higher performance than the conventional BTB even when BTB-X is given just half the storage budget of its conventional counterpart. For example, in Fig. 5a, the conventional BTB improves performance by 38% with a 21.5KB budget whereas BTB-X provides a 44% improvement with just 10.875KB of storage. The reason for this phenomenon is that BTB-X accommodates 2.8x more entries than a conventional BTB of equal storage budget; thus, halving BTB-X's budget still gives a capacity advantage over the conventional design.

Ignoring instruction prefetching and looking exclusively at performance gains stemming from reduced pipeline flushes, the trends are similar to above. For storage budgets of up to several tens of KBs, BTB-X outperforms a conventional BTB even with half of the latter's storage budget. For instance, Fig. 5a (blue segments of the bars) shows that BTB-X provides 13% gain with a 10.875KB budget whereas a conventional BTB with twice the budget (21.5KB) gains only 10%.

4.3 Impact of Tag Compression

For assessing the performance loss due to compressed tags, we compare the performance of BTB-X with 16-bit tags versus full tags for the smallest BTB size (10.875 KB). We focus on the smallest BTB as it is likely to suffer the highest degree of aliasing due to tag compression. Our results show that, full tags provide 38.21% performance gain, geo-mean across server and client traces, over the baseline compared to 38.16% with compressed tags, a difference of only 0.05%. This indicates that our tag compression scheme is able to preserve the entropy of higher-order bits.

5 Conclusion

The multi-megabyte instruction footprints of contemporary server applications cause frequent BTB and L1-I misses, which have become major performance limiters. Because BTB capacity greatly affects front-end performance in terms of flush rate and the efficacy of fetch-directed instruction prefetching, commercial products allocate tens to hundreds of KBs of storage to BTBs. To reduce the BTB storage requirements, this paper introduced an optimized BTB organization. The proposed design, BTB-X, leverages our insight that branch target offsets vary but tend to be much shorter than full target addresses. BTB-X uses an ensemble of five BTBs, each storing offsets of a different length, and also compresses the tags to track 2.8x more branches than a conventional BTB with an equal storage budget.

Acknowledgments

This work was supported in part by the Research Council of Norway (NFR) Under Grant 302279 to NTNU and by UK EPSRC Under Grant EP/M001202/1 to the University of Edinburgh.

References

[1] ChapSim. Accessed: Sep. 13, 2021. [Online]. Available: https://github.com/ChampSim/ChampSim
[2] NC State University, "The 1st instruction prefetching championship." Accessed: Sep. 13, 2021. [Online]. Available: https://research.ece.ncsu.edu/ipc/
[3] J. Bonanno *et al.*, "Two level bulk preload branch prediction," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Archit.*, 2013, pp. 71–82.
[4] B. Grayson *et al.*, "Evolution of the Samsung Exynos CPU microarchitecture," in *Proc. 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 40–51.
[5] Y. Ishii *et al.*, "Re-establishing fetch-directed instruction prefetching: An industry perspective," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.* 2021, pp. 172–182.
[6] R. Kumar *et al.*, "Boomerang: A metadata-free architecture for control flow delivery," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 493–504.
[7] R. Kumar *et al.*, "Blasting through the front-end bottleneck with shotgun," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 30–42.
[8] A. Pellegrini *et al.*, "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, Mar./Apr. 2020.
[9] G. Reinman *et al.*, "Fetch directed instruction prefetching," in *Proc. 32nd Annu. ACM/IEEE Int. Symp. Microarchit.*, 1999, pp. 16–27.
[10] A. Saporito, "The IBM z15 processor chip set," in *Proc. Hot Chips Symp.*, 2020, pp. 1–17.
[11] D. Suggs *et al.*, "The AMD "zen 2" processor," *IEEE Micro*, vol. 40, no. 2, pp. 45–52, Mar./Apr. 2020.