A Specialized BTB Organization for Servers

Truls Asheim
NTNU, Norway

Boris Grot
University of Edinburgh, UK

Rakesh Kumar
NTNU, Norway

1 INTRODUCTION AND MOTIVATION

Contemporary server applications feature massive instruction footprints stemming from deeply layered software stacks. These footprints 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 branches 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) [7, 8, 10], 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. Thus, IBM z-series processors [4], AMD Zen-2 [12], and ARM Neoverse N1 [9] feature 24K-entry, 8.5K-entry, and 6K-entry BTBs. With each BTB entry requiring 8 bytes or more, 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 [5]. Not only the BTB storage cost is high, it is increasing at a rapid pace. For example, the Samsung Exynos BTB storage budget increased nearly six fold (98.9KB to 561.5KB) from M2 to M6, over a period of about eight years [5]. While such massive BTBs are effective at capturing branch working sets, they do so at staggering area costs.

As the instruction footprints of server applications continue to expand, a trend also reflected in Google Web Search workload whose instruction footprint is growing at annualized rate of 27% [6], the BTB sizes and their storage overheads are destined to increase in future. Therefore, there is an urgent need to investigate storage-effective BTB organizations to combat the front-end bottleneck without necessitating prohibitive area budgets.

2 KEY INSIGHTS

We analyze conventional and state-of-the-art BTB organizations and observe that the branch targets are the single largest contributor to BTB storage cost. Further, we analyze the number of bits required for branch target *offsets* to assess if storing the offsets, instead of the full or compressed targets, can reduce BTB storage requirements.

Figure 1 plots the distribution of branch target offsets in the branch working sets of our workloads. The data includes both

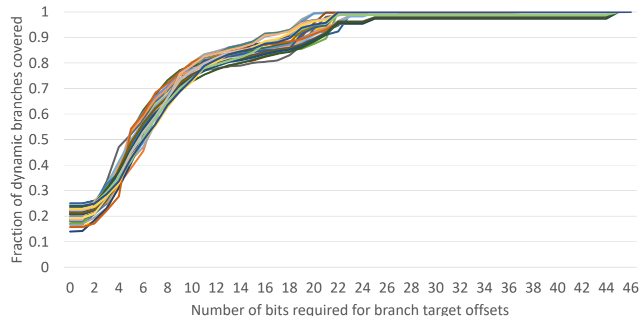

Figure 1: Distribution of branch target offsets.

conditional and unconditional branches; hence, it comprehensively covers the full branch working set. The X-axis shows the number of bits required to store offsets, while the Y-axis plots the fraction of dynamic branches covered.

As the figure shows, short offsets dominate the distribution with 54% of branches requiring only six bits or fewer for their offsets. A further 22% of branches only require between 7 and 10-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 [7]. 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 target offsets to short distances. Furthermore, return instructions get their target from the return address stack (RAS), thus they do not need to store any target bits in BTBs. Therefore, for the purpose of this analysis, we assume 0-bit offsets for return instructions.

Perhaps surprisingly, Figure 1 also shows that very few branches require a large number of bits for their offset. Indeed, a meagre 1% of branches requires more than 25 bits for their offsets. 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.

We gain two key insights from this analysis:

Key Insight 1: The targets of most branches lie relatively close in the virtual address space to the branch itself. As a result, storing the *distance* to the target, in the form of an offset from the branch instruction can provide drastic storage savings.

Key Insight 2: The target offset sizes are unevenly distributed with 0-6 bits, 7-10 bits, and 11-25 bits required to encode the offsets of 54%, 22% and 23% of branches respectively. Therefore, a single size offset field cannot provide storage optimal solution.

3 BTB-X

Based on these insights, we propose a new BTB design, called BTB-X, that stores target offsets instead of full or compressed target addresses. To accommodate the uneven distribution of target offsets, we size different ways of a set associative BTB-X to hold different

sized target offsets. A branch is allocated to a way whose offset field is at least as large as the number of bit required to store the target offset. We use an 8-way set associative BTB-X and leverage the data in Figure 1 to appropriately size the offset field of each way such that each way covers about 12.5% dynamically executed branches. Figure 1 shows that, on average, 0-, 4-, 5-, 7-, 9-, 11-, 19-, and 25-bit offsets cover about 20%, 36%, 46%, 61%, 72%, 79%, 90%, and 99% dynamic branches. Therefore, we size the 8-ways of BTB-X ways to hold 0-, 4-, 5-, 7-, 9-, 11-, 19-, and 25-bit target offsets respectively. Notice that about 20% of dynamic branches that require 0-bits for their offset are return instructions that read their target from RAS. Therefore, way-0 of BTB-X does not feature any storage for target offsets. Though return instruction do not get their target from BTB, they still need to be allocated BTB entries so that the branch prediction unit can identify them and pick their target from RAS while generating instruction stream to be fetched.

BTB-X covers 99% of the dynamically executed branches and we employ a very small conventional direct-mapped BTB, called BTB-XC, that stores full target addresses for the remaining 1% branches. Reserving a way in BTB-X for such branches would unnecessarily increase the storage requirements as these branches require much fewer entries than the number of sets in BTB-X. Indeed, based on our analysis, we size BTB-XC to store 64x fewer entries than BTB-X, i,e, 8x fewer entries than the number of sets in BTB-X.

4 EVALUATION

We use ChampSim [2], an open-source trace-driven simulator, to evaluate the efficacy of BTB-X on server workload traces provided for the first Instruction Prefetching Championship (IPC-1) [1]. The modeled processor resembles Intel Sunny Cove [3]. We compare the storage requirements and performance of BTB-X against a conventional BTB design (Conv-BTB) that stores full target addresses, and also against the state-of-the-art BTB design, called PDede [11], which stores compressed targets.

Storage comparison: Table 1 presents the number of branches the different BTB organizations (BTB-X, PDede, and Conv-BTB) can track at different storage budgets. The storage budgets shown are BTB-X storage required for storing 256, 512, 1K, 2K, 4K, 8K, and 16K branches. Our calculations assume a 48-bit virtual address space. As the table shows BTB-X stores significantly more branches than any other BTB organizations. Concretely, it stores 2.24x more branches than a conventional BTB organization. Compared to PDede, BTB-X stores 1.24x more branches at 0.9KB storage budget and 1.34x more branches at 58KB storage budget. BTB-X's advantage over PDede increases with storage budget because PDede entry size increases with the number of branches PDede can accommodate.

Performance comparison: Figure 2 presents the performance gains obtained by different BTB designs on a set of server workloads. The results are normalized to the performance of Conv-BTB with 0.9KB storage budget. Instruction prefetching (FDIP) is enabled in all designs including the baseline. As the figure shows BTB-X provides significantly higher performance than the Conv-BTB and PDede for equal storage budgets of up to 29KB and 14.5KB respectively. For instance, BTB-X provides 45% performance gain over the baseline compared to 38% and 27% of PDede and Conv-BTB, respectively, at 14.5KB 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 BTB-X and the other two designs diminishes. A key take-away from this figure is that BTB-X outperforms the conventional BTB even when it is given just half the storage budget of its conventional counterpart. For example, in Figure 2, the Conv-BTB improves performance by 27% with a 14.5KB budget whereas BTB-X provides a 31% improvement with just 7.25KB of storage. The reason for this behaviour is that BTB-X accommodates 2.24x more entries than Conv-BTB of equal storage budget; thus, halving BTB-X's budget still gives a slight capacity advantage over Conv-BTB.

Table 1: Number of branches in different BTB designs.

Storage	BTB-X (+ BTB-XC)	PDede	Conv-BTB
0.9KB	256 (+ 4)	210	116
1.8KB	512 (+ 8)	415	232
3.6KB	1K (+ 16)	820	464
7.25KB	2K (+ 32)	1617	928
14.5KB	4K (+ 64)	3190	1856
29KB	8K (+ 128)	6292	3712
58KB	16K (+ 256)	12405	7424

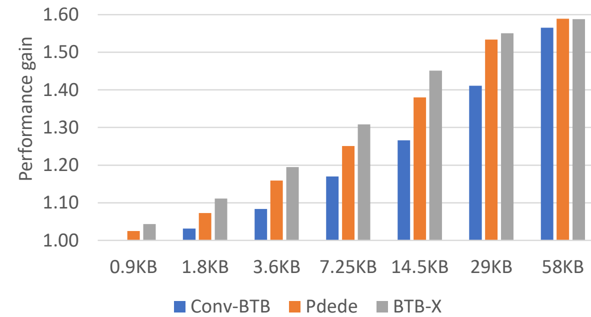

Figure 2: Performance comparison of different BTB designs.

5 CONCLUSION

As BTB capacity greatly affects front-end performance, commercial products allocate tens to hundreds of KBs of storage to BTBs. We propose a storage-effective BTB organization, called BTB-X, that stores target offsets in place of full target addresses and employs differently sized BTB-ways for storing offsets of different lengths, thus drastically reducing BTB storage requirements.

REFERENCES

[1] .. 1st Instruction Prefetching Championship. https://research.ece.ncsu.edu/ipc/.
[2] .. ChampSim Simulator. https://github.com/ChampSim/ChampSim.
[3] .. Ice Lake processors. https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/3.
[4] James Bonanno et al. 2013. Two level bulk preload branch prediction. In *HPCA'13*.
[5] B. Grayson et al. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *ISCA'20*.
[6] Svilen Kanev et al. 2015. Profiling a Warehouse-Scale Computer. In *ISCA'15*.
[7] Rakesh Kumar et al. 2017. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In *HPCA'17*.
[8] Rakesh Kumar et al. 2018. Blasting through the Front-End Bottleneck with Shotgun. In *ASPLOS'18*.
[9] Andrea Pellegrini et al. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020).
[10] G. Reinman et al. 1999. Fetch directed instruction prefetching. In *MICRO'99*.
[11] Niranjan K Soundararajan et al. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer. In *MICRO'21*.
[12] David Suggs et al. 2020. The AMD "Zen 2" Processor. *IEEE Micro* 40, 2 (2020).