Branch Target Buffer Organizations

Arthur Perais, Rami Sheikh

Branch Target Buffer Organizations

Arthur Perais
Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA
38000 Grenoble, France
arthur.perais@univ-grenoble-alpes.fr

Rami Sheikh
Arm
Raleigh, NC, USA
Rami.AlSheikh@arm.com

ABSTRACT

To accommodate very large instruction footprints, modern high-performance processors rely on fetch directed instruction prefetching through huge branch predictors and a hierarchy of Branch Target Buffers (BTBs). Recently, significant effort has been undertaken to reduce the footprint of each branch in the BTB, in order to either minimize the storage occupied by the BTB on die, or to increase the number of tracked branches at iso-storage. However, designing for branch density, while necessary, is only one dimension of BTB efficacy. In particular, BTB entry organization plays a significant role in improving instruction fetch throughput, which is a necessary step towards increased performance. In this paper, we first revisit the advantages and drawbacks of three classical BTB organizations in the context of multi-level BTB hierarchies. We then consider three possible improvements to increase the fetch PC throughput of the Region BTB and Block BTB organizations, bridging most of the performance gap with the impractical but highly storage-efficient Instruction BTB organization, thus paving the way for future very high fetch throughput machines.

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**.

KEYWORDS

Branch Target Buffers, BTB, Instruction fetch

ACM Reference Format:
Arthur Perais and Rami Sheikh. 2023. Branch Target Buffer Organizations. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3623774

1 INTRODUCTION

Code size has steadily increased over the years, especially in typical datacenter workloads (e.g., web server, databases, etc.) [19, 32]. This has caused a noticeable shift in hardware design philosophy since instruction caches, which were once large enough to contain most of the active code, are now direly under-provisioned for monolithic server-side workloads. Modern designs therefore rely on instruction prefetching to retrieve instructions from memory in advance.

Because instruction prefetching *is* control-flow speculation, many designs leverage the existing branch prediction infrastructure to perform fetch-directed instruction prefetching (FDIP) [47, 48]. A key aspect of FDIP is that while the instruction cache does not need to fit the entire code working set, the branch prediction structures, and especially the Branch Target Buffer [40] (BTB), do. In the CVP-1 [41, 43] 100M instruction traces used in this work, an average of 138KB is required to store the cache lines covering 90% of the dynamic instructions (resp. 319KB for 100%). This has caused BTBs to grow to hundreds of KBs [1, 23, 29]. For example, IBM's Telum core [29] stores more than 270k branch targets in the last level BTB. Such large structures have a significant silicon footprint. Therefore, several works have focused on improving information density in the BTB, both to reduce the number of entries as well as the bit count of each entry [4, 10, 24, 36, 38, 58].

Nevertheless, with or without density-improving techniques, BTBs with so many entries cannot provide 0-cycle taken branch turnaround, which is important to achieve high performance. In the CVP-1 traces used in this work, a 1-cycle taken branch penalty costs 0.8% geomean IPC (up to 2.2%) on an Intel Icelake-like configuration using a very large 512K-entry BTB. The reason is that high IPC code sections require high instruction fetch throughput to keep the backend fed, and the large cost of branch mispredictions makes restarting the pipeline a key performance limiter. BTB design plays a key role in both cases, and modern designs thus employ a hierarchy of BTB, where a small first level provides 0-cycle turnaround, while a second larger level incurs a taken branch penalty. Providing true 0-cycle turnaround severely limits the number of entries of the first level BTB and similarly works against the use of density improving techniques. For instance, indirection-based compression [53, 58] puts another table access on the next BTB index generation path, and encoding targets as offsets [4] puts an adder in the BTB index generation loop. In this paper, we choose to sidestep the question of actual storage requirements and to rather focus on BTB entry organizations and the properties they provide to facilitate or hinder high instruction fetch throughput. Concretely, we revisit three existing BTB organizations and highlight their respective advantages and drawbacks. We then argue that while highly efficient in the way it maps branch metadata to physical storage, the "classical" Instruction BTB organization where one entry tracks one branch is structurally not adapted to high performance designs. Unfortunately, we show that in a constrained setting, the other BTB organizations we cover, although more adapted to high performance designs as they tie branch information to instructions regions or blocks, do not achieve the same level of performance as Instruction BTB. As a result, we suggest microarchitectural improvements to bridge the gap with Instruction BTB while addressing its scalability challenges through tying entries to regions or blocks of instructions rather than individual branches.

*Grenoble Institute of Engineering

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3623774

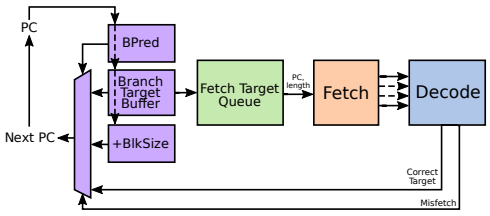


Figure 1: Decoupled PC generation flow using a BTB.

2 BACKGROUND

Taken branches are a well-known performance limiter in pipelined designs. The latency cost of such branches can be divided into a *direct cost* and an *opportunity cost*.

The former stems from the fact that the branch target is computed directly from the instruction bytes (register for indirect branches). The target is therefore available after c cycles, with c being the sum of the I-Cache access latency, decode latency and actual (direct) address computation latency. The *opportunity cost* is only relevant in superscalar designs. From the n (fetch width) instructions being fetched from the I-Cache, $n - x$ (with x in $\{1, n\}$) must be discarded if instruction x is a taken branch.

To address the *direct cost*, the Branch Target Buffer was proposed. The BTB is a cache for branch targets [40] that is accessed concurrently with the instruction cache, allowing fast branch target retrieval in the presence of a bigger and slower pipelined instruction cache. However, as the instruction footprint keeps growing, the BTB reach needs to extend accordingly, which inevitably leads to larger hence slower structures. To minimize the cost of taken branches, a hierarchy of BTBs can be considered [45]. This is the approach used in many modern designs [1, 8, 14–17, 23], although there remains recent designs where even hitting the first level BTB incurs bubble(s) on a taken branch [8, 11, 16, 29, 51].

A first key property of the BTB is that, like other branch prediction structures, it is not required to contain *correct* information. This is an important difference with the instruction cache and its sibling the trace cache [49], who must provide a consistent (and coherent, if the ISA mandates it) view of actual instructions stored in main memory. To the best of our knowledge, BTBs are not kept coherent with memory and must therefore employ a reactive scheme where predictions are checked e.g., by comparing them to information being fetched from the I-Cache [44, 47, 48]. This is the design we assume in the remainder of the paper.

The second key property of the BTB is that never taken conditional branches do not require BTB storage [9, 27, 28, 46]. In the CVP1 [43] server traces used in this work, an average of 34.8% of the dynamic branches are never taken conditional direct branches. This has implications on the management of the *Global Branch History* used by branch predictors. These implications are thoroughly discussed by Ishii et al. [27, 28].

2.1 Decoupled Fetching (DCF)

In older and/or less aggressive designs, the BTB and I-Cache are accessed concurrently with the current fetch PC. Therefore, if the I-Cache access misses, the whole fetcher is stalled. This is suboptimal because the branch prediction structures (predictors and BTB) are sufficient to autonomously generate fetch addresses: The I-Cache only serves the purpose of retrieving the actual instruction bytes to feed the rest of the pipeline. Consequently, Reinman et al. suggest *decoupling* fetch address generation from actual instruction retrieval with a queue [47]. This provides a natural instruction prefetching effect as multiple I-Cache misses can now be in flight concurrently.

A diagram of DCF is shown in Fig. 1: Each cycle, the PC generation stage enqueues into the Fetch Target Queue (FTQ) information that is later consumed by the Fetch stage to retrieve instructions bytes. The exact format of the information pushed to the FTQ depends on the BTB organization, but Fetch requires i) Where to start fetching and ii) How many instructions to fetch while Decode requires i) If instructions were predicted taken and if so ii) What the predicted targets are, to be able to validate them. In the remainder of this paper, we assume a decoupled fetcher organization as it is now the industry standard for high performance processors [1, 8, 14–17, 29]. Additional details on the pipeline stages implementing decoupled fetching in our simulation infrastructure are given in Section 4.

The main benefits of DCF come from the ability of the FTQ to fill up because of backpressure (e.g., narrow Fetch stage, I-Cache miss or full Reorder Buffer). First, as previously mentioned, multiple cache misses can overlap, which provides a prefetching effect. Second, the *direct* taken branch penalty can be hidden if there is enough backpressure. Third, in the presence of a bank-interleaved I-Cache, fetching past a taken branch by consuming two sequential FTQ entries that fall into different I-Cache interleaves becomes possible, mitigating the *opportunity* cost of taken branches. These benefits are structural and do not depend on the particular BTB entry organization, at least for those considered in this work. However, they rely on backpressure: fetching past a taken branch is not possible if the fetcher consumes FTQ blocks at the rate they are produced. The main drawback of this organization is the lengthening of the pipeline, which increases the misfetch[1] and the branch misprediction penalty.

2.2 Region BTB (R-BTB)

This BTB organization stores information for an aligned *region* of memory (e.g., 32B, 64B, etc.) in a single BTB entry [9, 23, 33]. This information includes metadata for several branches (referred to as *branch slots* in this work). The number of branch slots is decided at design time. If the region covered by an entry is a single instruction (e.g., 4 bytes in ARMv8 or RISC-V, 1 byte for x86), this organization degenerates into an Instruction BTB (I-BTB) in which each entry caches metadata for one branch. The R-BTB is accessed with a region aligned PC and provides i) How many sequential instructions to fetch in this region and ii) The type of branches to be fetched and if relevant, the target of the first taken branch (predicted taken or unconditional).

2.3 Block BTB (B-BTB)

Another BTB organization relies on the notion of *block*, which is a sequence of at most I instructions (or bytes), B of which can be *observed taken before* branches. *So far always taken* branches cause a block to end before I instructions. This is the definition used by

[1] BTB miss for a direct taken branch, which is resolved at Decode.

Amd in their recent designs [17]. The Block BTB is accessed using an instruction address and produces the number of instructions to fetch from that address as well as the next address to query the BTB with. If only one branch is tracked, then a BTB block is a basic block, as introduced by Yeh and Patt [63] and used with minor variations in [38, 39, 44, 46, 47]. A key detail is whether a *sometimes taken* conditional branch ends a block [63], or falls through until I instructions are reached [44] In this work, we assume the latter for our baseline, as it allows to compute the address of the sequential block in parallel with the BTB access, facilitating 0-cycle non taken branch penalty. The former trades off additional performance (Section 6.3) for storage as the fall-through target or block size has to be stored in the BTB entry.

3 CONTRASTING I-, R- AND B-BTB

3.1 Next PC Generation

To generate multiple fetch PCs[2] per cycle, the I-BTB needs to be accessed multiple times, which entails either multiporting, which is costly, or bank-interleaving, which still has timing overhead as information from 16 interleaves (in this work) need to be correctly arranged [13]. In the context of variable length ISAs, the indexes with which to access the different interleaves are unknown as they depend on instruction length. The index used to access the I-BTB must be aligned on the lowest instruction alignment, which is 1 byte in x86. Therefore, to cover the same number of bytes, x86 would require four times as many accesses as ARMv8 (4-byte instructions with 4-byte alignment). Conversely, R-BTB and B-BTB provide multiple fetch PCs from a single BTB entry, i.e., with a single access. We note, however, that none of these organizations are able to provide multiple basic blocks worth of fetch PCs with a single access. Since the average dynamic basic block size is 9.4 instructions in the studied traces, this suggests that single basic block throughput will limit overall performance in high IPC phases. In fact, even older organizations such as the Alpha EV8 [54] were able to provide two basic blocks each cycle, by relying on a fast next line predictor to provide the address of the second basic block to fetch. However, the EV8 did not use a BTB and would therefore fare poorly given modern codesizes, as it relied on a comparatively small I-Cache to store targets. Moreover the next line predictor already had limited accuracy at the time. This encourages us to focus on alternative BTB organizations that can support multiple block throughput.

3.2 Alignment

Each R-BTB entry caches information for an aligned region, which entails that a single access will only provide fetch PCs up to the end of the region. This is particularly limiting if the region size is small and suggests that to match B-BTB or I-BTB, the R-BTB should be dual-ported or interleaved in order to be able to generate instruction PCs from two sequential regions in the same cycle.

[2]In this context, a fetch PC is the PC of a single instruction, not the start PC of a sequence of instructions.

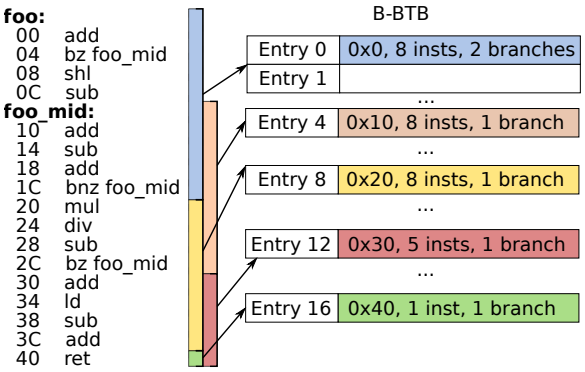


Figure 2: Redundancy in a 8-instruction Block BTB.

3.3 Tag Overhead

Assuming the same tag width in all organisations, I-BTB has the highest tag overhead because a single branch target is cached in each entry. R-BTB and B-BTB amortize the tag bits over multiple branch slots.

3.4 Redundancy

I-BTB and R-BTB cannot cache redundant information since each branch information may reside in at most one entry. Conversely, in B-BTB, multiple dynamic blocks may contain the same branch information, as shown in Fig. 2, for an 8-instruction B-BTB. The initial trace cache proposal [49] suffers from the same issue. The Figure first shows that Entry 0 and Entry 4 overlap. Both track branch 0x1C because label *foo_mid* is both the fall-through of *foo* and the target of branches 0x04, 0x1C and 0x2C. Redundancy is not limited to two entries. For instance, if 0x4 was the target of a branch, then branch 0x1C would be tracked by three entries (0 from 0x0 to 0x1C, 1 from 0x4 to 0x20, and 4 from 0x10 to 0x2C).

Redundancy will propagate until the next unconditional (or always taken conditional) branch. For instance, if any instruction from 0x10 to 0x1C were an unconditional branch, then Entry 0 and Entry 4 would both stop at that instruction and be followed by the same next BTB entry. However, because 0x1C is not unconditional (assumed not always taken in this example), a new block has to be allocated as the fall-through of Entry 0. This block, tracked by Entry 8, overlaps with Entry 4 and both track branch 0x2C. This pattern continues as no instruction between 0x20 and 0x2C and 0x30 and 0x3C are unconditional (or always taken conditional) branches, causing the allocation of Entry 12 and Entry 16. Both entries terminate at the unconditional branch 0x40, forcing the "synonym" paths to merge and preventing further redundancy. Generally, any two overlapping BTB blocks that are not terminated by an unconditional (or always taken conditional) branch will cause the allocation of two fall-through blocks that are also overlapping.

The only structural mitigation to these phenomena is when there are no branches to cache in the blocks, since only blocks with at least taken once branches allocate entries in the BTB. To prevent "synonym" paths, blocks can be forced to terminate at region boundaries (e.g., 64B). However, this defeats the purpose of B-BTB as it prevents generating fetch PCs from different regions

in a single access while not even entirely preventing redundancy: Entry 0 would still overlap with Entry 4 if blocks were terminated at 32B boundaries (0x1C).

3.5 Allocation Behavior

Taken branches always allocate in the BTB, regardless of the organization. The main difference between I-BTB and both R-BTB and B-BTB is that in I-BTB, allocating a new branch displaces another one. This is not necessarily the case in R- and B-BTB: if there exists an entry tracking the block with a free branch slot, no other branch gets displaced.

While seemingly an advantage for R- and B-BTB, this actually makes these organizations much less agile. Indeed, since not all instructions are branches, it would be hugely inefficient to provision as many branch slots per entry as the maximum number of instructions covered by a block or region. As a result, a region or block with more taken branches than number of branch slots will see branches competing for storage, leading to *BTB hit branch slot miss* cases, which have the same penalty as regular BTB misses. Conversely, blocks or regions with fewer branches than branch slots in each BTB entry will make poor use of the available storage.

This limitation can be addressed in two fashions: First, by increasing the number of branch slots per entry, at the cost of increasing inefficiency for regions that have few taken branches. Second, by providing decoupled but shared "overflow" storage to the BTB entries, allowing to dynamically extend the number of branch slots as needed. This is the path chosen by some industry designs e.g., IBM z16 [29], AMD Bobcat [11], Samsung Exynos [23], as well as Confluence [33]. "Overflow" branches incur extra latency [11, 23]. For B-BTB, an alternative is to keep the number of branch slots limited but proactively split entries to prevent a new taken branch from displacing the metadata pertaining to another branch, at the cost of increased pressure on the BTB. This is one of the improvements we study in Section 6.3.

3.6 BTB Hierarchy

So far, the discussion has focused on the functional behavior of three different BTB organizations. However, similar to caches, modern processors employ a hierarchy of BTBs, with lower level BTBs being faster at the cost of being smaller, and higher level BTBs being bigger at the cost of being slower e.g. [8, 14–16, 23, 29, 51]. Organizing BTBs in a hierarchy was first suggested by Perleberg and Smith [45].

3.6.1 Fast Next BTB Index Generation. If one aims to fully hide the taken branch penalty, the BTB should be able to provide the next BTB index in a single cycle. In other words, BTB indexing, tag check, data access and processing (e.g., selecting which targets to use based on branch predictions), as well as capturing this next BTB index in the flip-flop that drives BTB access should fit in one cycle. One might argue that the average IPC observed on real machines is generally well below the maximum IPC achievable by said machines. As a result, there will generally be backpressure and the taken branch penalty will often be hidden through queuing, hence there is no need to spend circuit design effort on a fast first level BTB. Such a design philosophy would overlook the fact that providing 0-cycle turnaround remains valuable to maximize the pipeline refill rate on pipeline flushes, as well as high IPC phases. In our limit study using the processor configuration depicted in Table 1 but a very large 512K-entry I-BTB, a 1-cycle taken branch penalty reduces the geomean IPC by 0.8% (up to 2.2%) compared to no penalty.

Providing 0-cycle turnaround puts stringent limits on the BTB size as well as the amount of entry "post processing" that can be done. Density techniques such as encoding the target as an offset to the entry PC and compressing tags may not be viable given the latency constraint. This naturally leads to a tiered BTB organization where the lower level is kept small and gives up on the density techniques mentioned above to limit access latency. Conversely, higher levels tradeoff access time for significantly higher density that comes from encoded and/or compressed target and tag representation, deduplication, and indirections [53, 58]. Consequently, minimizing the time required to provide the next BTB index is comparatively more important than improving density for the lower level BTBs, and especially the first level BTB. To further illustrate this concern, we provide two examples.

First, consider that in R-BTB, the next BTB index is not just a selection between the targets in the branch slots and the fall-through based on the branch predictions. Rather, the next BTB index also depends on what is the unaligned fetch PC the entry is being accessed through. For instance, assume a 32B-region entry that has two branch slots caching information for a branch at 0x4 and at 0x1C, respectively. If the entry is accessed through PC 0x0, then the possible next BTB indices are i) The target of 0x4 ii) The target of 0x1C and iii) The region fall-through. Conversely, if the entry is accessed through PC 0x10, then only two next BTB indices are possible i) The target of 0x1C and ii) The region fall-through. In other words, the offset of each branch slot must be compared to the unaligned PC used to access the entry, and the result participates in target selection. Assuming the branch predictor is the critical path, this adds at least one AND gate to the critical path: each branch prediction is now AND'ed with the result of the comparison before driving the selection lines of the next fetch PC multiplexer. If the BTB access is already the critical path, then the offset comparison is added to it. While this may be inconsequential depending on how much slack is available, neither the I-BTB nor the B-BTB have to perform such an offset comparison.

The second example relates to the number of accesses that need to be performed to provide multiple fetch PCs in a single cycle. Ideally, the first level BTB must remain small to meet timing. While current commercially available AMD designs use a 1K+-entry L1BTB incurring 0 bubbles[3] on taken branches (except calls) [14, 17], the previous generation design did use a tiny L0BTB with only eight entries to provide 0-cycle turnaround. This L0BTB was backed up by a 256-entry L1BTB (1-cycle penalty) and a 4K-entry L2BTB (4-cycle penalty) [16]. In both cases, we understand the BTBs to be B-BTBs that feature 2 branch slots. However, to fill both slots, the branches must belong to the same cache line. The Samsung Exynos line of processors also used three levels of BTBs with two levels of 128B R-BTBs with 8 branch slots in each entry and a 0-cycle turnaround "graph based" μBTB, which we understand to be an embodiment of the B-BTB organization with a specific training algorithm.

[3]The low latency combined to the structure size suggests the use of ahead pipelining [55].

The number of fetch PCs per access is of particular interest if a non ahead pipelined design were chosen for the first level BTB, i.e., true 0-cycle taken branch penalty on L1BTB hit. To achieve this, the number of entries would be limited in order to meet timing. As a result, the L1BTB would likely be built as a fully associative structure to maximize the amount of code patterns that can be cached concurrently. If this BTB is organized as an I-BTB, however, this entails multiple associative searches per cycle, hence multiple search ports. As a result, I-BTB is not well adapted for a small first level BTB, even in the context of a fixed length ISA. Conversely, R-BTB or B-BTB can provide many fetch PCs with a single access, although to maximize performance, implementing two search ports would be needed to handle code blocks spanning two consecutive regions in R-BTB. Set interleaving is also an option although it will likely suffer from imbalance if the first level BTB is small. To summarize, we argue that for such designs, the B-BTB organization is better suited for the first level BTB.

3.6.2 Heterogeneous BTB Hierarchies. Although this work focuses on homogeneous BTB hierarchies, we note that the organization best suited for the L2 and L3 BTBs may not be the one most suited for the L1 BTB. Due to information redundancy in the B-BTB organization, a given B-BTB level may require more entries to store the same number of distinct branches compared to an R-BTB. In absolute terms, this means that choosing the B-BTB organization for large BTBs can waste a significant amount of storage. This hints that a heterogeneous hierarchy may conceptually be advantageous (this was previously suggested by Perleberg and Smith [45], although in a context where the BTB also provides direction predictions). We leave the study of heterogeneous hierarchies for future work.

4 EXPERIMENTAL FRAMEWORK

This work first aims to compare existing BTB organizations in terms of fetch PC throughput. While the different organizations can be swapped in a straightforward manner, it is not obvious what a fair comparison is, since the different organizations have different storage and design tradeoffs. Moreover, techniques such as tag compression using an indirection [53, 58] will mechanically favor organizations that have fewer tags per targets, i.e., R-BTB and B-BTB. As a result, we choose to sidestep the question of storage and to keep the number of cacheable branch targets the same across organizations. This favors the I-BTB organization as it does not tie multiple branch slots to a region or a block.

4.1 Simulator

In this study, we use the ChampSim [22] cycle-level, trace-based simulator. The pipeline was modified to model a decoupled fetcher using two levels of BTBs. All changes are added on top of the *master* branch of the ChampSim repository, commit *29f568c*. The frontend pipeline stages are depicted in Fig. 3. Note that in the examples depicting L2BTB hit, the fetch PC misses L1BTB, but this does not prevent sequential instructions (or region or block) from moving on to Fetch speculatively: There are no bubbles between BP and FTQ for the missing fetch PC. When the L2BTB hit is resolved, any over-fetch is corrected. The next fetch PC, however, does observe the taken branch penalty. Our experiments are performed with

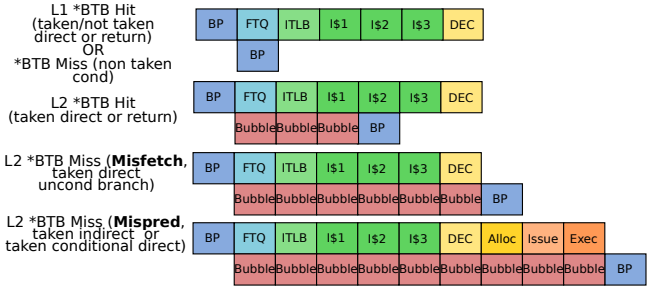


Figure 3: Frontend pipeline timing diagram of the decoupled fetcher organization considered in this work. All indirect branches except *returns* incur an additional 1-cycle bubble.

immediate update of the BTBs and branch predictors. Moreover, the latency to fill (resp. evict) entries from higher (resp. to higher) BTB levels is not modeled. The FTQ is bypassed when empty.

We further augment Fetch by allowing it to read multiple FTQ entries each cycle if they map to different I-Cache set interleaves. In other words, all organizations can fetch any 16 instructions across any number of taken branches from at most 8 different cache lines if i) The FTQ has information about these instructions ii) The cache lines map to different I-Cache set interleaves iii) The decode queue has space. The branch predictor can provide as many branch predictions per cycle as required by the specific BTB organization.

We consider a processor model with an aggressive superscalar width of 16. Since the average dynamic basic block size is 9.4 in the traces, fetching across a taken branch is necessary to achieve ideal throughput. Table 1 summarizes the main simulation parameters.

4.2 Workloads

In this study, we use "secret" server traces from the First Championship Value Prediction (CVP-1) [41, 43]. The traces were generated from ARMv8 datacenter class workload binaries. We first consider a subset of traces that have > 1 I-Cache misses per kilo instruction as our goal is to be representative of workloads with large instruction footprints. Of this subset of 783 traces, we use only 147, as most of the traces are sensitive to the accuracy with which ARM-specific addressing modes are modeled, and support for pre/post increment indexing is currently not present in ChampSim. We therefore focus on the traces that are not sensitive[4] to this simulator abstraction. The pipeline structures are warmed up for 50M instructions, and statistics are collected during the next 50M instructions.

5 GAUGING THE POTENTIAL OF I-BTB, R-BTB AND B-BTB

This initial experiment aims to delineate the potential of the three organizations, assuming the processor depicted in Table 1 and a huge (8K sets, 32-ways) BTB that has a 0-cycle taken branch penalty. We harmonize the width of the organizations by considering R- and B-BTB with 16-instruction regions/blocks (64B) and I-BTB with 16 banks. A larger region size (128B) was considered but proved significantly less appealing for R-BTB in realistic configurations:

[4]IPC does not vary by more than 5% when addressing mode support is enabled in the CVP-2 [42] infrastructure.

Table 1: ChampSim Pipeline Configuration

Branch Prediction	∘ 64KB Hashed Perceptron (16 4K-entry tables, 0-232 bit histories 8-bit weights, ChampSim commit *29f568c*) [30, 60], as many prediction per cycle as needed per cycle ∘ 64-entry Return Address Stack (RAS) [31] ∘ 4K-entry gshare-like indirect target predictor
BTB Sizes	∘ ideal: 512K-entry BTB, 0c bubble* ∘ real: 3K-entry L1BTB: 512 sets/6 ways, full tag, LRU, 0c bubble* ∘ real: 13K-entry L2BTB: 1024 sets/13 ways, full tag, LRU, 3c bubble* *Non-return indirect branches cause an additional bubble
I-BTB R-BTB B-BTB	∘ 1 Br/entry ∘ 64B regions, *n* Br/entry ∘ 64B blocks, *n* Br/entry, one block per entry
Fetch Target Queue	∘ 64-entry, one entry relates to a single cache line
Frontend	∘ 16-wide Fetch, 64-entry Decode Queue ∘ 16-wide Decode, 64-entry Allocate Queue ∘ 16-wide Allocate
Backend	∘ 352-entry ROB, 128-entry LQ, 72-entry SQ, 128-entry IQ ∘ 16-wide execute (11 Misc. + 3 Ld + 2 St) ∘ 16-wide Commit
Memory	∘ 32KB L1I$: (64 sets/8 ways), 8-way set interleaved, 3c lat., 16 MSHRs, LRU ∘ 48KB L1D$: (64 sets/12 ways) 5c ld-use, 16 MSHRs, LRU, IPStride prefetcher ∘ 512KB L2: (1024 sets/8 ways), 15c load to use., 32 MSHRs, LRU, NextLine prefetcher ∘ 2MB LLC: (2048 sets/16 ways), 35c load to use., 64 MSHRs, LRU ∘ 64-entry ITLB: (32 sets/4 ways), 1c lat. ∘ 64-entry DTLB: (32 sets/4ways), 1c lat. ∘ 1536-entry L2TLB: (128 sets/12 ways), 8c lat. ∘ 32GB DRAM: 3200 MHz, quad-channel

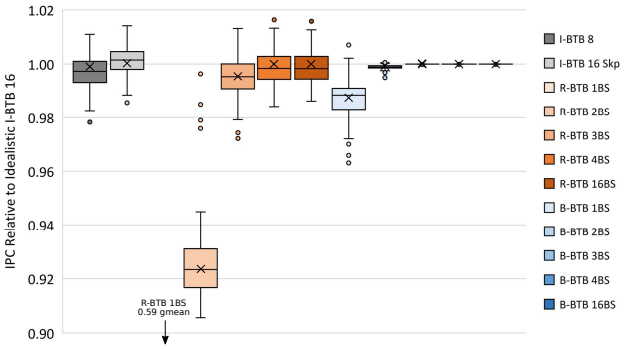


Figure 4: Performance of different BTB organizations relative to I-BTB 16. Idealistic 512K-entry L1BTB, CVP-1 server traces. The cross is the geometric mean, the box horizontal lines are 1^{st} quartile, median and 3^{rd} quartile. Whiskers span any point within $Q1-(1.5\times(Q3-Q1))$ (resp. $Q3+(1.5\times(Q3-Q1))$). Dots are any point that are further than $1.5\times(Q3-Q1)$ from Q1 and Q3 respectively.

As the region size grows, the number of branch slots per entry must grow accordingly, implying structures with fewer entries at iso-branch slots. B-BTB, however, can benefit from larger blocks without increasing the number of branch slots. In fact, 128B and 256B regions with the same number of branch slots as 64B regions provide slight performance advantages in specific configurations, as we will show in further experiments.

Fig. 4 reports whisker plots of the IPC achieved by the simulated processor given the different BTB organizations, relative to I-BTB 16.[5] As we consider a huge BTB that covers the whole code footprint, the performance is limited by the structural constraints of the different BTB organizations during phases where the backend does not apply backpressure, such as pipeline fills. The first two whiskers relate to I-BTB that can only provide up to 8 fetch PCs per cycle (*I-BTB 8*) and I-BTB that always provides 16 fetch PCs per access regardless of taken branches (*I-BTB 16 Skp*). These configurations are used to gauge performance sensitivity to fetch PC throughput of the selected trace on the simulated microarchitecture. The key takeaway is that once a high enough fetch PC throughput it attained, additional throughput does not provide significant performance improvement, unless the pipeline flushes often, as we will show in Section 6.5.2. In this context, "high enough" relates to the ILP that the backend is able to extract as providing more fetch PC throughput than usable ILP eventually leads to microarchitectural instruction buffers filling up. Specifically, while *I-BTB 8* degrades IPC by up to 2.2% (0.2% geomean), *I-BTB Skp* improves it only by up to 1.4% (0.1% geomean). This is despite the large difference in average number of fetch PCs generated per access: 5.6 (*I-BTB 8*), 7.7 (*I-BTB 16*) and 15.9 (*I-BTB 16 Skp*).

For R- and B-BTB, a first structural limitation is the number of branch slots per region or block: In R-BTB and B-BTB, the IPC diminishes with the number of branch slots per entry because additional misfetches and mispredictions are caused by untracked branches. In this dataset, 2 branch slots per entry are sufficient for B-BTB, but R-BTB still sees slight improvement when going to 4 and even 16 branch slots. However, the average occupancy of the branch slots (sampled on the entire BTB structure every 1M instructions) is only 1.60 (16-slot R-BTB) and 1.06 (16-slot B-BTB): Provisioning 2 branch slots per entry is storage inefficient for both organizations.

The second structural limitation is that R-BTB cannot generate fetch PCs past the region boundary. Therefore, even with 16 branch slots per entry and –close to– no misses, it is not able to always keep up with I-BTB and B-BTB. In this idealistic case (huge BTB and 16 branch slots), each I-BTB and B-BTB accesses generate an average of around 7.7 fetch PCs, while each R-BTB access generates only an average of 6.2, translating to up to 1.4% IPC loss (0.2% geomean).

We further investigate metadata redundancy within the L1 B-BTB. To gather this information, we inspect the BTB every 1M simulated instructions. For each branch PC tracked by at least one BTB entry, we count the total number of entries tracking that PC. I-BTB and R-BTB have a ratio of 1, while B-BTB has a ratio of 1.06 entries on average, indicating that 6% of the branch slots are lost to redundancy. The ratio vary only slightly as the number of branch slots changes. One could argue that the BTB is quite large (512K entries), such that some entries may remain lingering forever without being used, artificially increasing the redundancy ratio. However, we found comparable ratios when more reasonably sized structures were used. Nevertheless, this suggests that to attain the same reach as I-BTB and R-BTB in the non-ideal case, the B-BTB should have more branch slots in total.

[5]All performance numbers reported in this paper are normalized to 512K-entry I-BTB 16 to keep the baseline consistent

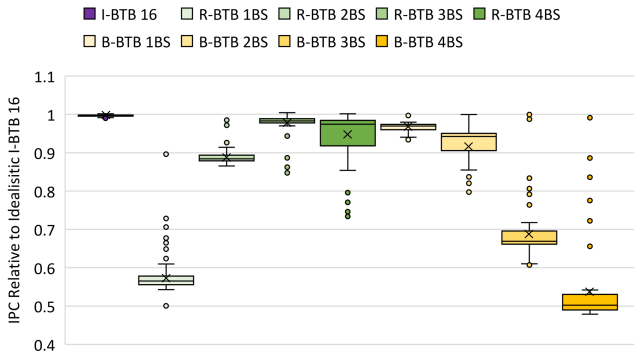


Figure 5: IPC of realistic I-, R- and B-BTB configurations normalized to the IPC of idealistic I-BTB 16.

6 BUILDING AN EFFICIENT BTB HIERARCHY

The next step of our analysis is to determine if the performance advantage of B-BTB over R-BTB endures the switch to a realistic BTB hierarchy. Further experiments use the BTB sizes reported in Table 1. Although the first level BTB is a large structure, it is heavily contented: On the traces used in this study, the 3K-entry L1 I-BTB (resp. 13K-entry L2 I-BTB) has an average hitrate[6] of 76.3% (resp. 99.9%).

6.1 Performance of Realistic BTB Hierarchies

We study a 3K-entry (512 sets/6 ways) L1 I-BTB with 0-cycle taken branch penalty, backed-up by a 13K-entry L2 I-BTB (1024 sets/13 ways) with a 3-cycle taken branch penalty. These sizes reflect the number of branch slots in Amd Zen 4 [17]. For R-BTB and B-BTB, the structures are resized to account for the fact that each entry provisions 1 (1x I-BTB), 2 (0.5x I-BTB), 3 (1K-entry 256 sets/4 ways L1BTB, 4.5K-entry 256 sets/18 ways L2BTB) or 4 (0.25x I-BTB) branch slots. Fig. 5 reports the IPC of realistic I-, R- and B-BTB normalized to idealistic (512K-entry) I-BTB 16.

First, the different organizations behave differently as the number of branch slots per entry changes. With a single branch slot per entry, R-BTB behaves poorly as cache lines generally feature more than one taken branch. B-BTB performs comparatively well, closer to the realistic I-BTB organization (1.74 vs. 1.79 geomean IPC). The IPC difference can be attributed to redundancy as well as misfetches and mispredictions caused by untracked branches. A branch is cached in 1.04 (L1) and 1.05 (L2) entries on average for 1 branch slot per entry, translating to lower hitrates than I-BTB: 60.8% vs. 76.3% (L1) and 97.8% vs. 99.9% (L2), leading to more misfetches and mispredictions. This compounds with blocks having multiple taken branches that yield *BTB hit branch slot misses*. Combined branch mispredictions and misfetches per kilo-instructions is 5.91 MPKI compared to only 0.84 for realistic I-BTB.

Second, using more branch slots to prevent misfetches and mispredictions caused by untracked branches is detrimental for B-BTB. Instead of branches contending for branch slots within entries, blocks now contend for entries. The trend applies for R-BTB after 3 branch slots per entry, although the performance decrease is largely flatter. The reason for the different slope is that several blocks can be cached in a single R-BTB entry, amortizing the reduced number of entries.

The main conclusion we draw from the Figure is that in a context where the BTB is oversubscribed, making good use of branch slots is paramount. In terms of branch slot utilization, the most efficient R- and B-BTB configurations are those with a single branch slot per entry. However, they are not able to achieve the geomean IPC of the realistic I-BTB configuration (*I-BTB 16*). For R-BTB, 3 slots per entry actually achieve the highest IPC, despite suboptimal branch slot use (1.57/1.55 slot used per L1/L2BTB entry, on average). In the next Section, we introduce possible techniques to bridge the remaining gap, which are necessary to provide high performance as a wide I-BTB organization requires the number of BTB banks to scale with the expected branch prediction width.

6.2 Even/Odd Set Interleaving the L1 R-BTB

The main structural limitation of R-BTB is that it cannot generate fetch PCs past a region boundary. To address this limitation, the structure can be set-interleaved. This is a known technique [13, 46] that permits generating up to 32 fetch PCs (from 16) from two consecutive regions with single ported arrays. The cost of doing so is additional hardware to rearrange fetch PCs (even or odd interleave information first) as well as doubling the number of tag checks and branch predictions each cycle. We refer to this configuration as *2L1 R-BTB*. In designs featuring a small, fully associative L1BTB, interleaving entries across two banks when only tens of entries are available is likely to suffer from imbalanced use of the two banks. As a result, those designs would have to rely on dual porting the L1BTB, which is costlier than interleaving. Regardless, this design hides latency only if both sequential entries are found in the L1BTB during lookup. Furthermore, it still does not permit generating fetch PCs over a taken branch.

6.3 Splitting Entries in B-BTB

In the baseline B-BTB, or in the R-BTB, when all branch slots of an entry are used, a newly taken branch has to overwrite existing information to be tracked. While many replacement policies can be devised (LRU, unconditional direct first, etc.), the key is that information is lost, potentially causing a misfetch or a misprediction down the line. Alternatively, some form of supplemental branch slots can be dynamically allocated to entries [11, 23, 29, 33], at a latency cost.

However, with B-BTB, there exists a third option, used in [44, 46, 63]. An entry with n branch slots needing to cache $n+1$ branches can be split into two distinct entries. Conceptually, the update logic may insert the new branch in a staging block local to the update logic that has $n+1$ slots, and then split the entry at the instruction following the n^{th} branch slot, assuming branch slots are ordered by offset of the branch in the block. By construction, the n^{th} slot in the staging block is a *sometimes taken* conditional branch.

A drawback of splitting is that a split entry does not have a default fall-through anymore. Typically, with 16-instruction blocks and 4-byte instructions, the fall-through of the current entry can be computed by adding 64 to the current access PC in parallel with the BTB access. However, when a block is split, the fall-through

[6]Taken branches hitting the BTB over total taken branches.

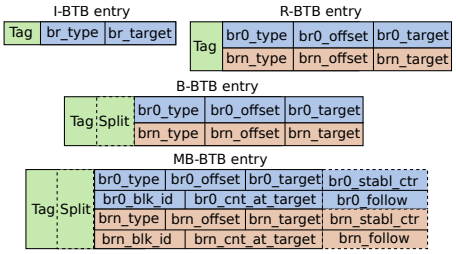


Figure 6: Possible entry layout of MB-BTB, contrasted against I-, R- and B-BTB. Dashed boxes are optional fields depending on required B/MB-BTB features.

depends on the block offset of the last branch slot, hence it cannot be computed in parallel with the BTB access. If this addition prevents the first level BTB access from meeting timing, split entries will incur a bubble. To avoid paying this cost for non-split entries, a bit may be added in each entry to inform whether an entry is split or not.

6.4 MultiBlock BTB

6.4.1 Leveraging Unconditional Control Flow. As previously discussed in Section 2, the BTB contains speculative information that is eventually checked against the ground truth: instruction bytes for direct branches or calculated targets for indirect branches. This key property allows organizing the BTB as a *MultiBlock BTB* (MB-BTB) where each entry caches information about a chain of blocks, much like the *trace cache* [49], without suffering from one of its main drawbacks: Maintaining coherence with the rest of the cache hierarchy. Regardless of whether this is done in hardware (e.g., x86) or software (e.g., ARMv8), this is highly problematic for a trace cache as a single instruction may be part of many traces, hence many entries. As a result, if a cached instruction is modified, the whole trace cache must be searched and all entries containing that instruction must be invalidated. This has led to trace cache implementations such as the Pentium 4's, where writes to instruction memory could cause the invalidation of the whole trace cache, notably lowering the performance of self-modifying code [26]. Since the BTB is speculative, it is not problematic for it to cache incorrect information. Therefore, fully leveraging the block organization and making it multiblock will not require hardware to support writes to instruction memory.

The key idea behind MB-BTB is that all B-BTB entries that terminate with an unconditional direct branch are always followed by the same entry. Therefore, instead of occupying a second entry, the block at the target of the unconditional direct branch can be "pulled" into the entry tracking the unconditional branch. Up to $\#BranchSlots + 1$ blocks can be chained in this fashion (although the last block cannot be terminated by a taken branch). This is the approach used in Amd Zen 4 (2 branch slots), with two limitations: direct calls are not eligible and the second branch must be in the 64-byte aligned region containing the target of the first branch [17].

A possible layout for an MB-BTB entry is provided in Fig. 6 and contrasted against I-, R- and B-BTB. First, each branch slot is associated with the block they belong to through the *br_blk_id* field. Second, each branch that pulls its target block in needs to provide the instruction count of said block. This is done through the *br_cnt_at_target* field. Lastly, optional fields can be added: *br_follow* and *br_stabl_ctr* are provided to enable always taken (and almost always taken) conditional branches and indirect branches with a single target to pull their target block in.

For instance, consider an MB-BTB entry with 2 branch slots, containing 2 unconditional direct branches. This means that the entry may cache up to three blocks. The *br_offset* field (blue) of the first unconditional direct branch informs of the number of bytes in the first block. The *br0_cnt_at_target* field (blue) informs of the number of bytes between the target of the first unconditional branch (*br0_target*) and the second unconditional branch (i.e., the second block of the entry). Finally, the *br1_cnt_at_target* field (orange) informs of the number of bytes between the target of the second unconditional branch (*br1_target*) and the end of the block (i.e., the third block of the entry). MB-BTB entries may also be split if needed, which requires the *brX_offset* fields to be actual offsets of the branches in their respective blocks (given by *brX_blk_id*). MB-BTB suffers from the same drawback as B-BTB with entry splitting: Entry information is needed to compute the fall-through if a block has been split at a sometimes taken conditional branch.

6.4.2 Categorizing Conditional and Indirect Branches. While we initially consider MB-BTB for direct unconditional branches only (including calls), the same idea can be used for always –or almost always– taken conditional branches as well as indirect branches that always jump to the same target. Those respectively represent 15.0% and 9.1% of the dynamic branches on average in CVP-1 server traces [43]. However, after a taken conditional branch or indirect branch is first encountered, its future behavior is unknown. Therefore, it may not be ideal to immediately pull the target block of this branch into the MB-BTB entry. Rather, this should be done only after the behavior repeats enough times. In this work, we experimented with several thresholds and found that immediately pulling the target of conditional branches while requiring the same target 63 times in a row for indirect branches works well enough. This improves density while limiting the number of times a multiblock entry is quickly broken down because e.g., a branch thought to be always have the same target changes target. Note that for conditional branches, implicit filtering is already applied since a conditional branch that is not taken at the time the BTB block is allocated will not be always taken, and is therefore not eligible for pulling its target into the block.

Practically, we keep one 6-bit counter per branch slot in the MB-BTB entries, *brX_stabl_ctr* in Fig. 6. This counter is incremented at update time if the branch is indirect and the target matches the one in the MB-BTB. If not, the counter is reset. The counter is always kept at the threshold and *brX_follow* remains set for unconditional direct branches and always taken conditional branches. Once the counter reaches the threshold for an indirect branch, the corresponding *brX_follow* bit is set, and the fall-through of the branch (if present) is replaced by the target block during block update. Should the branch change behavior, the bit and counter would be reset, and the target block replaced by the fall-through (direct branch) or removed (indirect branch). Note that counters may only be implemented in the L1BTB while other levels only record the *brn_follow* bit. In a hierarchy with an L1BTB with only few tens

of entries [15, 16, 44], this would greatly limit the overall storage footprint required to pull the target blocks of indirect branches. Further tuning of the heuristic determining which conditional and indirect branches are eligible is left for future work.

We also found that there is a slight performance advantage in disallowing the last[7] branch of a block from pulling its target. This is advantageous due to the tendency of (M)B-BTB to cache redundant information, as was discussed in 3.4. For instance, if there are two call sites for a function, and the two call instructions occupy the last branch slots of their respective MB-BTB entry, then allowing them to pull their target may lead to two different fallthrough blocks. Conversely, preventing them from pulling their target will ensure both will share the same successor block, reducing redundancy. The same applies to other types of eligible branches.

6.4.3 Entry Update and Splitting Entries. Splits in MB-BTB work like in B-BTB. Updates are also similar, although the update logic needs to be augmented to handle conditional or indirect branches that reach or leave the "pull target" threshold.

Interestingly, when an always taken conditional branch has pulled its target information but becomes not taken, there is a choice to make: Should the MB-BTB entry be amended so that the target block of the branch and any subsequent blocks are removed from the entry, or should it be kept as is? In theory, this choice boils down to whether the branch will be mostly taken or not. Indeed, by "pulling" the target block of a conditional branch into the MB-BTB entry of the branch, we have introduced a *non-taken* branch penalty. Reversing the behavior of the branch to that of a normal conditional branch in the BTB entry would make that *non-taken* penalty disappear but would reintroduce the *taken* branch penalty. As a result, in practice, it is not obvious whether the correct choice is to keep the entry as is or to update it, especially if the first level BTB hides the penalty in both cases. In this work, we immediately downgrade the always taken branch to a normal conditional branch and remove its target block (and potential followers) from the entry. The same applies to indirect branches that were previously treated as always jumping to the same target.

6.5 Impact of R- and B-BTB Improvements on Performance

6.5.1 R-BTB. Fig. 7 presents the gain brought by accessing two sequential R-BTB entries through interleaving (*R-BTB 2L1 2/3BS*), along with two R-BTB configurations that have the number of sets and ways (i.e., geometry) depicted in Section 6.1 for 2/3 branch slots per entry, but actually feature 16 branch slots per entry (*R-BTB 2/3Geo 16BS*), as well as four R-BTB configurations with 128B regions with 2, 3, 4 and 6 branch slots per entries (*R-BTB 128B 2/3/4/6BS*).

The first takeaway is that being able to get fetch PCs from two sequential entries has limited impact, providing up to 1.4% (0.5% geomean) IPC improvement with 2 branch slots per entry, and up to 1.2% (0.2% geomean) with 3 branch slots, over similar configurations with a single interleave.

The second takeaway is that the performance degradation of R-BTB compared to I-BTB comes from pressure on branch slots

[7]Branch slot n if entries feature n branch slots.

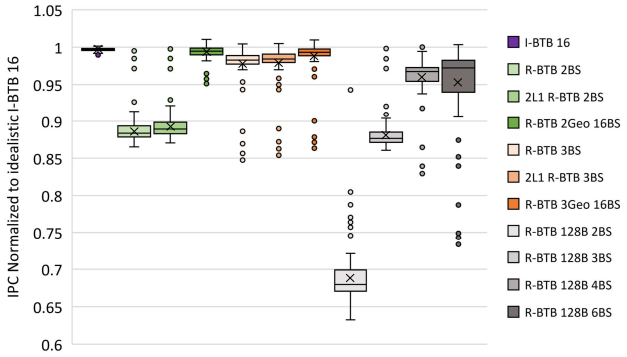


Figure 7: IPC of R-BTB, interleaved R-BTB and 128B region R-BTB configurations normalized to the IPC of idealistic I-BTB 16.

rather than entries, at least for 2 and 3 branch slots per entry. Indeed, by keeping the same number of entries but providing 16 branch slots, performance significantly increases from 1.60/1.76 (*2BS/3BS*) geomean IPC to 1.79/1.78 (*2Geo 16BS/3Geo 16BS*), with realistic I-BTB 16 at 1.79. This could be seen as an upper bound of the performance achievable with shared overflow branch slots for regions that have more than 2/3 taken branches. Nevertheless, once the number of entries gets low enough, e.g., 4BS geometry with 16 slots per entry (not shown in the Figure), the trend inverts and BTB contention starts becoming the major bottleneck.

The last takeaway is that increasing the region size does not provide significant performance advantage for R-BTB, although the average number of fetch PCs generated by access increases from 6.2 (*R-BTB 2/3BS*) to 6.8 (*2L1 R-BTB 2BS*) and 6.7 (*2L1 R-BTB 3BS*), and finally 7.4 (*R-BTB 128B 4/6 BS*). However, while this is a clear win for *128B R-BTB 4BS* over *R-BTB 2BS*, the reduction in number of BTB entries in *128B R-BTB 6BS* causes additional misfetches and branch mispredictions due to BTB misses, such that *2L1 R-BTB 3BS* remains the best performer of the realistic R-BTB configurations. Much like when the region size is 64B, at iso-branch slots, there is an optimal number of branch slots per entry that balances BTB misses with misfetches/mispredictions caused by untracked branches. From the Figure, that number is 3 for 64B regions, and 4 for 128B regions, given the workloads and BTB sizes considered in this work.

6.5.2 B-BTB and MB-BTB. Fig. 8 shows the IPC of the various B-BTB proposals, normalized to that of an idealized I-BTB 16, for realistically sized structures. The first two boxes relate to the realistic I-BTB 16 configuration, and the best realistic R-BTB configuration from Fig. 7. The Figure then presents various B- and MB-BTB configurations with 1, 2 and 3 branch slots per entry. For MB-BTB, *UncndDir* means that only target blocks of unconditional direct branches (excluding calls) are pulled, *CallDir* adds direct calls, and *AllBr* adds conditional branches and indirect branches that met the conditions discussed in Section 6.4.2 (63).

In general, R-BTB and MB-BTB still lag behind I-BTB due to the heavily contended L1BTB. In fact, the highest performing MB-BTB scheme (*MB-BTB 2BS AllBr*) is behind *B-BTB 1BS Splt* at 1.74 vs. 1.78 geomean IPC, suggesting that in a context where the L1BTB

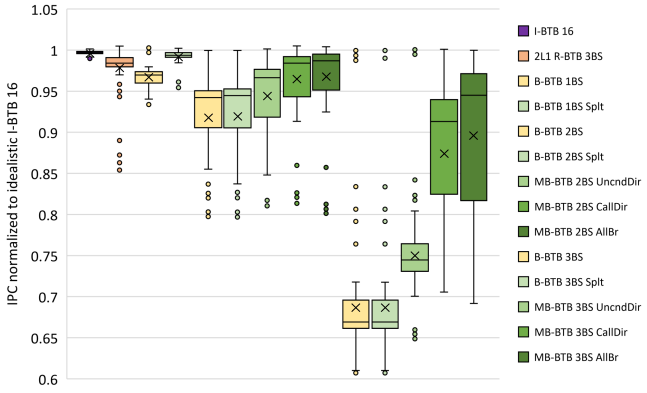


Figure 8: IPC normalized to ideal I-BTB 16 for the proposed B-BTB improvements and MB-BTB.

is contented, more entries featuring a single branch slot is the best tradeoff, as long as entries can be split. Indeed, for 1 branch slot, splitting brings 2.6% geomean IPC improvement, from 1.75 to 1.78. I-BTB 16 achieves 1.79 geomean IPC, while the idealistic I-BTB 16 configuration achieves 1.80 (0.3% speedup). Despite being considered in [44], splitting is not needed with 2 and 3 branch slots as branches do not compete for branch slots within blocks as much, enabling parallel fall-through computation and BTB access.

Nevertheless, with more than one branch slot per entry, the relative potential of MB-BTB is quite high, especially when the target of unconditional direct branches and calls are pulled into their predecessor entries. Pulling call targets is comparatively more important: for 2 (resp. 3) branch slots, pulling targets of unconditional direct branches increases performance by 2.7% (resp. 9.1%) geomean, and pulling targets of direct calls further increases performance by 2.2% (resp. 16.5%) geomean. Pulling the target of always taken conditional branches as well as indirect branches with a single target in their predecessor block has a more modest impact: up to 1.8% (0.3% geomean) for *MB-BTB 2BS AllBr* and up to 4.1% (2.6% geomean) for *MB-BTB 3BS AllBr*, over similar *MB-BTB CallDir* configurations.

A last improvement avenue for B- and MB-BTB specifically is that contrarily to R-BTB, the reach (i.e., block size) of an entry can be increased without necessarily having to increase the number of branch slots in each entry, as entry splitting can gracefully handle supernumerary taken branches. Conversely, in R-BTB, increasing entry reach generally requires increasing the number of branch slots per entry, as previously shown in Fig. 5, and does not provide any performance advantage vs. a cache line size region (64B) in our experiments. In I-BTB, wider fetch PC generation requires additional banks.

In this experiment, we therefore consider the highest performing B-BTB (*B-BTB 1BS Splt*) and MB-BTB configurations (*MB-BTB 2/3BS AllBr*) and extend entry reach without changing the number of branch slots per entry. Fig. 9 reports the IPC normalized to the idealistic I-BTB 16 configuration. For *B-BTB 1BS Splt* with a single branch slot, we see negligible increase when moving from 16-instruction (*B-BTB 16 1BS Splt*) blocks to 32-instruction blocks (*B-BTB 32 1BS Splt*). Further block size increase similarly did not yield improvement.

For *MB-BTB 2BS AllBr*, increasing reach from 16 to 32 instructions has a noticeable impact on performance (up to 6.3% speedup, 1.3% geomean), while increasing from 32 to 64 has negligible impact.

Finally, for *MB-BTB 3BS AllBr*, the impact of larger reach is more significant. Although configurations with 3 branch slots per entry have higher performance variability, using 64-instruction blocks provides 1.73 geomean IPC while 16-instruction blocks only provides 1.61 IPC, which represent a 6.8% geomean IPC increase. Note however that baseline B-BTB configurations with increased entry reach remain at a comparable level of performance as their lower reach counterparts, simply because the higher reach is often unused due to an unconditional branch terminating the block.

Fig. 10 summarizes the number of average fetch PCs generated per BTB access along geomean IPC for the realistic configurations. Overall, MB-BTB is very efficient at improving block utilization, which noticeably improves performance compared to plain B-BTB when each entry provisions 2 and 3 branch slots. This improvement comes from MB-BTB being able to partially compensate for B-BTB misses by providing multiple blocks on hits. However, in a constrained setting, and given the workloads used in this study, this does not translate to competitive performance and a simple B-BTB with a single branch slot per block and entry splitting is the best tradeoff. Even if R-BTB does not suffer from metadata duplication, its best configuration, *2L1 R-BTB 3BS*, still lags behind *B-BTB 1BS Splt* (1.4% higher geomean IPC). Nonetheless, we argue that MB-BTB delivers as expected in terms of fetch PCs per access. Indeed, MB-BTB can only improve performance over the idealistic I-BTB if redundancy is absorbed by additional entries and the backend is swift enough to expose the frontend bottleneck, which happens in two cases: i) Program sections where extractible ILP is higher than the dynamic basic block size (9.4 on average, in this work) and ii) The pipeline often has to refill e.g. because of branch or memory dependency mispredictions. Unfortunately, neither of these cases are observed in our experimental framework, especially as branch MPKI is only 0.84 on average (0.15 minimum, 3.55 maximum, 0.72 median) and ChampSim implements oracle memory dependency prediction.

To illustrate this unrealized potential, we consider two final experiments. In the first, we use the idealistic 512K-entry *I-BTB 16* and *MB-BTB 64 AllBr* in combination with an ideal backend that is limited only by ILP within an 8K instruction window.[8] Figure 11a depicts the performance advantage of *MB-BTB 64 AllBr* over *I-BTB 16* along the average dynamic basic block size for all traces. Speedups are significant (13.4% geomean, up to 15.6%) and correlate well with the dynamic basic block size: since smaller blocks cannot efficiently use the bandwidth of *I-BTB 16* (or even *B-BTB 16*), the highest speedups are obtained for smaller block sizes. Conversely, as the average basic block size grows towards 16, the improvement diminishes, although the minimum speedup is still significant at 6.0%. This confirms that MB-BTB does have a fundamental advantage over I-BTB and B-BTB, but that this advantage cannot be realized in current designs where the backend cannot extract enough of the underlying ILP.

[8]All data dependencies are enforced but all instructions require a single cycle to execute, with functional units always available. The whole window can be executed in a single cycle if dependencies permit it. The whole window can be retired in a single cycle.

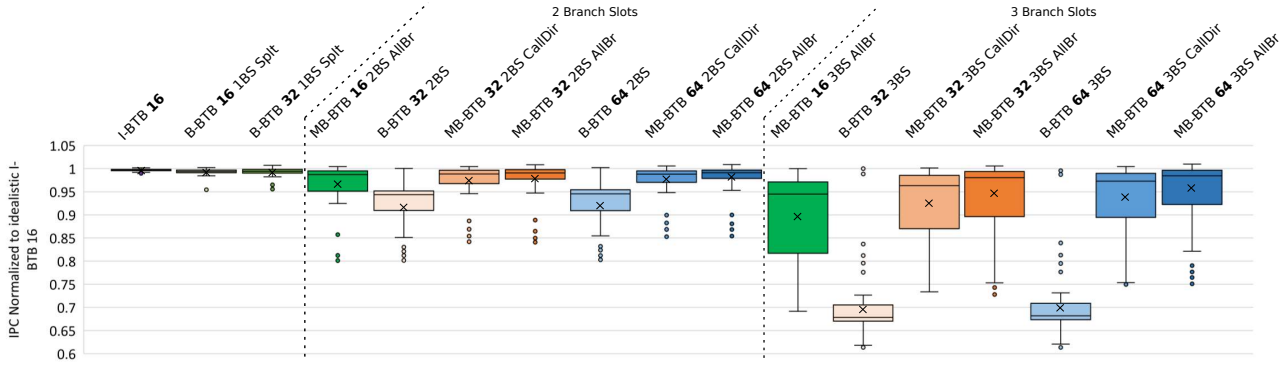


Figure 9: IPC normalized to ideal I-BTB 16 for B- and MB-BTB when increasing block size.

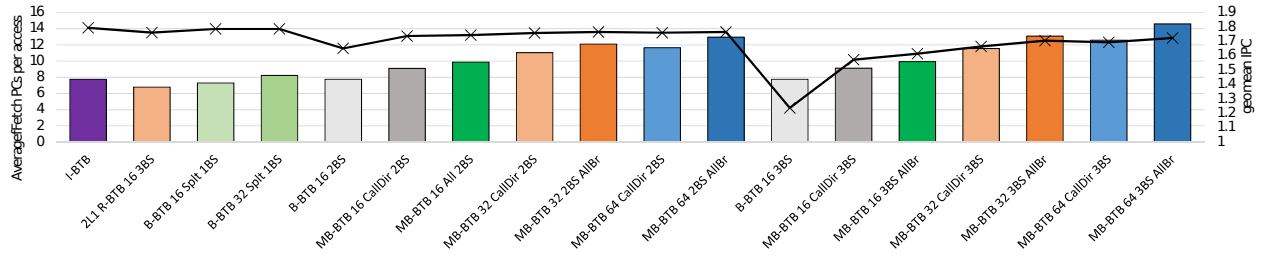


Figure 10: Average Fetch PCs provided by each BTB access and geomean IPC of various realistic BTB configurations.

Nevertheless, MB-BTB can still provide an advantage over I-BTB and B-BTB with a realistic backend when programs show poor behavior in terms of branch –and memory dependency– predictability. Indeed, our second experiment increases branch MPKI by progressively reducing the size of the conditional branch predictor, and showcases that the performance uplift brought by *MB-BTB 64 AllBr* directly correlates with branch MPKI, as illustrated in Figure 11b. While artificial, this last experiment showcases that workloads with high branch MPKI can benefit from MB-BTB on pipeline refills, even if the backend will eventually throttles the frontend.

7 RELATED WORK

7.1 BTB Density

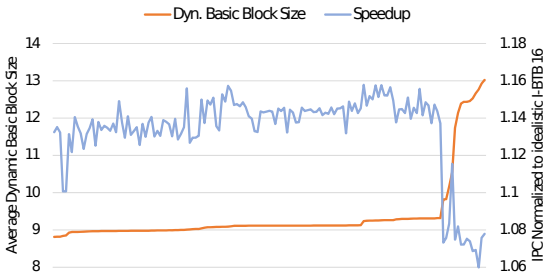
Yeh and Patt first consolidate basic block information in a BTB entry [63], with improvements proposed by Reinman et al. [46] in the *Fetch Target Buffer*. In both cases, an entry can track one taken branch, limiting taken branch throughput.

PDede restructures the BTB to reduce redundancy through decomposing the target address into region, page, and offset pieces that are stored in different tables (address partitioning and deduplication) and optimized for the common case where the branch and its target are in the same page [58]. Tag compression and address partitioning were first proposed for the BTB in [53]. Kobayashi et al. propose an alternative to tag compression where high order bits of the target are taken from the access PC when the branch jumps within a specific region of code (*near* branch) [36]. An embodiment of this scheme is implemented in the SPARC T4 [56]. Another variation proposed by Hoogerbrugge is to combine or mix entries being able to store *near* or *far* targets in the same structure [25]. BTB-X similarly stores *offsets* defined as bits to be concatenated to significant bits of the branch PC to form the address, rather than performing an addition. The BTB has multiple ways, each able to store a different offset size [4, 5]. Fagin suggests using partial tags in the BTB as full tags are not cost effective [18]. Gupta et al. [24] store branch targets as offsets only in the last level BTB and recommend the usage of skewed indexing [52]. Other works focus on BTB replacement policies, either purely in hardware [2] or through profiling and software hints [57]. Those techniques are generally orthogonal to MB-BTB.

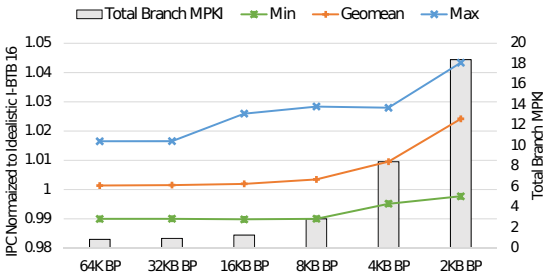
7.2 Instruction Prefetching

Reinman et al. first introduced decoupled fetching (DCF) that decouples branch prediction from instruction fetch using a queue that stores the addresses to be fetched [46–48]. This allows branch prediction to progress beyond instruction cache misses, exploiting instruction memory level parallelism and performing Fetch Directed Instruction Prefetching (FDIP). Ishii et al. revisit DCF within modern processors and provide possible optimizations for global history register management and early resteer on BTB misses [27, 28].

Confluence [33] recognizes that the I-Cache and FDIP leverage the same metadata. It uses an R-BTB operating at the cache line granularity and uses the same prefetch metadata to prefetch into both the BTB and I-Cache using record-and-replay (temporal streaming). Shotgun [38] observes that server workloads have global control flow that consists of unconditional branches that jump between



(a) Speedup of *MB-BTB 64 AllBr* over *I-BTB 16* when an ideal backend is used. Traces are sorted from shortest to longest average dynamic basic block size.



(b) Min/Avg/Max speedup of *MB-BTB 64 AllBr* over *I-BTB 16* as the branch predictor size is decreased, thereby increasing average branch MPKI.

Figure 11: Limit study of the potential of *MB-BTB 64 AllBr* over *I-BTB 16*. 512K-entry BTBs.

regions of code, and local control flow with high spatial locality within each region, allowing for dense encoding. The BTB is thus statically partitioned among the two to prevent collisions, and the cache lines accessed for a given region are recorded and used to issue prefetches. Shotgun performs instruction prefetching both through the FTQ and through the dedicated information stored in the BTB. Using a dedicated BTB for unconditional branches was first proposed by Yeh and Patt [63]. Other instruction prefetching work perform instruction prefetching through a dedicated structure [3, 20, 21, 37, 59, 64] or through profiling and software prefetching [6]. These works focus on improving fetch latency by maximizing I-Cache hits, but do not tackle multiple block fetch.

7.3 BTB Prefetching

Boomerang performs BTB prefetching by decoding the data returned from the L2 on an I-Cache miss to proactively fill the BTB [39]. Twig [35] profiles execution to identify critical BTB misses offline and then inserts new BTB prefetch instructions into an optimized version of the binary. The IBMz12 core performs L2 to L1BTB prefetching ("preloading") upon encountering both an L1 BTB miss and I-Cache miss. In that event, the L2BTB is searched to provide the branch metadata for the whole 4KB code region to be loaded into the L1BTB [8]. BTB prefetching can also be added on top of MB-BTB, although decode-based prefetching may not always be able to chain blocks, e.g. if the target of the prefetched block is already in the BTB.

7.4 Fetching Multiple Blocks per Cycle

The works most closely related to MB-BTB are *Zero-bubble Always Taken* and *Zero-bubble Often Taken* (Samsung's M5 [23]), and the Amd Zen 4 BTB [17]. The former embeds the target of always or mostly taken branches in their predecessor BTB entries. This mechanism was only leveraged to remove the taken branch penalty in a non-decoupled design, and not to fetch multiple blocks each cycle as it did not consider embedding more than one additional target in the BTB entry. A similar effect can be achieved with a *Branch Target Instruction Cache* (BTIC) that stitches instructions at the target of a branch with said branch *after* Fetch [34]. Regardless, it is unclear if indirect branches were considered in the M5 mechanism, when single target indirect branches make up an average of 9.1% of the dynamic branches in the CVP-1 server traces [43]. The Zen 4 BTB, while even closer to the MB-BTB proposal, does not consider "pulling" the target entry of single target indirect branches as well as always taken conditional branches when they represent 9.1% and 15.0% of the dynamic branches, respectively.

A second closely related work is the Branch Address Cache (BAC) [61, 62]. Each BAC entry captures a subtree of the application's control flow graph (CFG), with each tree node corresponding to a branch. Each node stores both the branch target and the fall-through address, leading to significant duplication as the subtree depth increases. The depth depends on the number of branches predicted per cycle, with said predictions selecting the path to be followed in the subtree, thereby generating multiple fetch blocks each cycle. An MB-BTB entry can similarly provide the addresses of multiple fetch blocks but does so by storing only one path of the CFG.

Rotenberg et al. introduce the Trace cache, which records traces made of multiple basic blocks [49, 50]. Like BAC, a trace is selected using branch predictions. Like MB-BTB, each entry only caches a single path in the CFG. Trace caches suffer from redundancy of information, for which mitigations have been proposed [7], but more importantly, from costly self-modifying code and invalidation flows as invalidated instructions may reside in many entries [26]. The BTB, need not invalidate entries when instruction memory is written.

Lastly, the Alpha EV8 can fetch up to two basic blocks each cycle using a fast next line predictor [12] backed up by a complex conditional branch predictor [54] and the I-Cache itself. This design completely sidesteps the BTB but already suffered from limited next line predictor accuracy even when the typical code footprint was much smaller compared to that of modern datacenter class workloads.

8 CONCLUSION

This work surveys and compares three BTB organizations in the context of a BTB hierarchy and large instruction footprint workloads. We argue that Instruction BTB is not adapted to high performance designs as it requires scaling the number of interleaves with the fetch width. Unfortunately, more amenable organizations, Region and Block BTB, are not able to provide the same level of

performance as I-BTB when realistically sized structures are used. This paper considered one improvement to R-BTB and two improvements to B-BTB to bridge the gap. We found experimentally that in a realistic setting, avoiding BTB misses is comparatively more important than increasing the number of fetch PCs per access. This makes B-BTB with a single branch slot per entry and entry splitting the best performing practical organization. However, we showed that MB-BTB significantly increases the number of fetch PCs per BTB accesses, such that if the code footprint fits in the BTB, the MultiBlock BTB can bring back the performance level to that of I-BTB, and even modestly outperform it, on a subset of server traces from CVP-1 [41, 43].

REFERENCES

[1] Narasimha Adiga, James Bonanno, Adam Collura, Matthias Heizmann, Brian R. Prasky, and Anthony Saporito. 2020. The IBM z15 High Frequency Mainframe Branch Predictor Industrial Product. In *Proceedings of the International Symposium on Computer Architecture*. 27–39. https://doi.org/10.1109/ISCA45697.2020.00014

[2] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A. Jiménez. 2018. Exploring Predictive Replacement Policies for Instruction Cache and Branch Target Buffer. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 519–532.

[3] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. Divide and Conquer Frontend Bottleneck. In *Proceedings of the International Symposium on Computer Architecture*. IEEE, 65–78.

[4] Truls Asheim, Boris Grot, and Rakesh Kumar. 2021. BTB-X: A Storage-Effective BTB Organization. *IEEE Computer Architecture Letters* 20, 2 (2021), 134–137.

[5] Truls Asheim, Boris Grot, and Rakesh Kumar. 2023. A Storage-Effective BTB Organization for Servers. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 1153–1167.

[6] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. 2019. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *Proceedings of the International Symposium on Computer Architecture*. 462–473.

[7] Bryan Black, Bohuslav Rychlik, and John Paul Shen. 1999. The Block-based Trace Cache. In *Proceedings of the International Symposium on Computer Architecture*. 196–207.

[8] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. 2013. Two Level Bulk Preload Branch Prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 71–82.

[9] Brian K. Bray and Michael J. Flynn. 1991. Strategies for Branch Target Buffers. In *Proceedings of the International Symposium on Microarchitecture*. 42–50.

[10] Ioana Burcea and Andreas Moshovos. 2009. Phantom-BTB: a Virtualized Branch Target Buffer Design. In *Proceeding of the International Symposium on Architectural Support for Programing Languages and Operating Systems*. 313–324.

[11] Brad Burgess, Brad Cohen, Marvin Denman, Jim Dundas, David Kaplan, and Jeff Rupley. 2011. Bobcat: AMD's low-power x86 processor. *IEEE Micro* 31, 2 (2011), 16–25.

[12] Brad Calder and Dirk Grunwald. 1995. Next Cache Line and Set Prediction. In *Proceedings of the International Symposium on Computer Architecture*. 287–296.

[13] Thomas M. Conte, Kishore N. Menezes, Patrick M. Mills, and Burzin A. Patel. 1995. Optimization of Instruction Fetch Mechanisms for High Issue Rates. In *Proceedings of the International Symposium on Computer Architecture*. 333–344.

[14] Advanced Micro Devices. 2020. Software Optimization Guide for AMD EPYC™ 7003 Processors, Pub 56665, Rev 3. , 28-29 pages. [Online; accessed Nov.-2022].

[15] Advanced Micro Devices. 2020. Software Optimization Guide for AMD Family 15h Processors, Pub 47114, Rev 3.08. , 35 pages. [Online; accessed Nov.-2022].

[16] Advanced Micro Devices. 2021. Software Optimization Guide for AMD Family 17h Processors, Pub 55723, Rev 3.00. , 27 pages. [Online; accessed Nov.-2022].

[17] Advanced Micro Devices. 2023. Software Optimization Guide for AMD Zen4 Microarchitecture, Pub 57647, Rev 1. , 21 pages. [Online; accessed Apr.-2023].

[18] Barry Fagin. 1997. Partial Resolution in Branch Target Buffers. *IEEE Trans. Comput.* 46, 10 (1997), 1142–1145.

[19] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceeding of the International Symposium on Architectural Support for Programing Languages and Operating Systems*. 37–48.

[20] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive Instruction Fetch. In *Proceedings of the International Symposium on Microarchitecture*. 152–162.

[21] Michael Ferdman, Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal Instruction Fetch Streaming. In *Proceedings of the International Symposium on Microarchitecture*. 1–10.

[22] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. *The Championship Simulator: Architectural Simulation for Education and Competition*. https://doi.org/10.48550/ARXIV.2210.14324

[23] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *Proceedings of the International Symposium on Computer Architecture*. 40–51. https://doi.org/10.1109/ISCA45697.2020.00015

[24] Vishal Gupta and Biswabandan Panda. 2022. Micro BTB: a High Performance and Storage Efficient Last-level Branch Target Buffer for Servers. In *Proceedings of the International Conference on Computing Frontiers*. 12–20.

[25] Jan Hoogerbrugge. 2000. Cost-efficient Branch Target Buffers. In *European Conference on Parallel Processing*. 950–959.

[26] Intel. 2022. Intel® 64 and IA-32 Architectures Optimization Reference Manual, Section 3.6.8.1. www.intel.com.

[27] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2020. Rebasing Instruction Prefetching: An Industry Perspective. *IEEE Computer Architecture Letters* 19, 2 (2020), 147–150.

[28] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. 172–182.

[29] Christian Jacobi. 2021. *The > 5GHz Next Generation IBM Z Processor Chip*. https://research.ibm.com/publications/the-greater5ghz-next-generation-ibm-z-processor-chip

[30] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic Branch Prediction with Perceptrons. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 197–206.

[31] David R. Kaeli and Philip G. Emma. 1991. Branch History Table Prediction of Moving Target Branches due to Subroutine Returns. In *Proceedings of the International Symposium on Computer Architecture*. 34–42.

[32] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the International Symposium on Computer Architecture*. 158–169.

[33] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: Unified Instruction Supply for Scale-out Servers. In *Proceedings of the International Symposium on Microarchitecture*. 166–177.

[34] A. Richard Kennedy, Mike Alexander, Eric Fiene, Jose Lyon, Belli Kuttanna, Rajesh Patel, Mydung Pham, Michael Putrino, Cody Croxton, Suzanne Litch, et al. 1997. A G3 PowerPC/sup TM/superscalar low-power microprocessor. In *Proceedings of COMPCON, Digest of Papers*. 315–324.

[35] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A. Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-guided BTB Prefetching for Data Center Applications. In *Proceedings of the International Symposium on Microarchitecture*. 816–829.

[36] Ryotaro Kobayashi, Yuji Yamada, Hideki Ando, and Toshio Shimada. 1999. A cost-effective Branch Target Buffer with a two-level Table Organization. In *Proceedings of the International Symposium of Low-Power and High-Speed Chips*.

[37] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. 2013. RDIP: Return-address-stack Directed Instruction Prefetching. In *Proceedings of the International Symposium on Microarchitecture*. 260–271.

[38] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting Through the Front-end Bottleneck with Shotgun. In *Proceeding of the International Symposium on Architectural Support for Programing Languages and Operating Systems*. 30–42.

[39] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A Metadata-free Architecture for Control Flow Delivery. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 493–504.

[40] Johnny K. F. Lee and Alan Jay Smith. 1984. Branch Prediction Strategies and Branch Target Buffer Design. *Computer* 17, 01 (1984), 6–22.

[41] Arthur Perais. 2018. *First Championship Value Prediction Secret Traces*. https://doi.org/10.18709/perscido.2023.02.ds384

[42] Arthur Perais, Rami Sheikh, Milind A. Choudhary, Chris Wilkerson, and Alaa R. Alameldeen. 2021. *Second Championship Value Prediction*. https://microarch.org/cvp1/cvp2/rules.html

[43] Arthur Perais, Rami Sheikh, Eric Rotenberg, Vinesh Srinivasan, Mikko Lipasti, Chris Wilkerson, and Alaa R. Alameldeen. 2018. *First Championship Value Prediction*. https://www.microarch.org/cvp1/cvp1online/contestants.html

[44] Arthur Perais, Rami Sheikh, Luke Yen, Michael McIlvaine, and Robert D. Clancy. 2019. Elastic Instruction Fetching. In *Proceedings of the International Symposium on High Performance Computer Architecture*. IEEE, 478–490.

[45] Chris H. Perleberg and Alan Jay Smith. 1993. Branch Target Buffer Design and Optimization. *IEEE transactions on computers* 42, 4 (1993), 396–412.

[46] Glenn Reinman, Todd Austin, and Brad Calder. 1999. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Proceedings of the International Symposium on Computer Architecture* (Atlanta, Georgia, USA). 234–245. https://doi.org/10.1145/300979.300999

[47] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch Directed Instruction Prefetching. In *Proceedings of the International Symposium on Microarchitecture*. 16–27.

[48] Glenn Reinman, Brad Calder, and Todd Austin. 2001. Optimizations Enabled by a Decoupled Front-end Architecture. *IEEE Trans. Comput.* 50, 4 (2001), 338–355.

[49] Eric Rotenberg, Steve Bennett, and James E. Smith. 1996. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the International Symposium on Microarchitecture*. 24–34.

[50] Eric Rotenberg, Steve Bennett, and James E. Smith. 1999. A Trace Cache Microarchitecture and Evaluation. *IEEE Trans. Comput.* 48, 2 (1999), 111–120.

[51] Anthony Saporito. 2020. The IBM z15 Processor Chip Set. In *Proceedings of the Hot Chips Symposium*. 1–17. https://doi.org/10.1109/HCS49909.2020.9220508

[52] André Seznec. 1993. A Case For Two-way Skewed-associative Caches. In *Proceedings of the International Symposium on Computer Architecture*. 169–170.

[53] André Seznec. 1996. Don't Use the Page Number, but a Pointer to it. In *Proceedings of the International Symposium on Computer Architecture*. 104–104.

[54] André Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. 2002. Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor. In *Proceedings of the International Symposium on Computer Architecture*. 295–306.

[55] André Seznec and Antony Fraboulet. 2003. Effective Ahead Pipelining of Instruction Block Address Generation. In *Proceedings of the International Symposium on Computer Architecture*. 241–252.

[56] Manish Shah, Robert Golla, Gregory Grohoski, Paul Jordan, Jama Barreh, Jeffrey Brooks, Mark Greenberg, Gideon Levinsky, Mark Luttrell, Christopher Olson, Zeid Samoail, Matt Smittle, and Thomas Ziaja. 2012. Sparc T4: A Dynamically Threaded Server-on-a-Chip. *IEEE Micro* 32, 2 (2012), 8–19. https://doi.org/10.1109/MM.2012.1

[57] Shixin Song, Tanvir Ahmed Khan, Sara Mahdizadeh Shahri, Akshitha Sriraman, Niranjan K Soundararajan, Sreenivas Subramoney, Daniel A. Jiménez, Heiner Litz, and Baris Kasikci. 2022. Thermometer: profile-guided btb replacement for data center applications. In *Proceedings of the Annual International Symposium on Computer Architecture*. 742–756.

[58] Niranjan K. Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. Pdede: Partitioned, Deduplicated, Delta Branch Target Buffer. In *Proceedings of the International Symposium on Microarchitecture*. 779–791.

[59] Viji Srinivasan, Edward S. Davidson, Gary S. Tyson, Mark J. Charney, and Thomas R. Puzak. 2001. Branch History Guided Instruction Prefetching. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. 291–300.

[60] David Tarjan and Kevin Skadron. 2005. Merging Path and gshare Indexing in Perceptron Branch Prediction. *ACM transactions on architecture and code optimization* 2, 3 (2005), 280–300.

[61] Tse-Yu Yeh, Deborah T Marr, and Yale N. Patt. 1993. Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proceedings of the International Conference on Supercomputing*. 67–76.

[62] Tse-Yu Yeh, Deborah T. Marr, and Yale N. Patt. 2014. Author Retrospective for Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache. In *Proceedings of the International Conference on Supercomputing, 25th Anniversary Volume*. 24–25.

[63] Tse-Yu Yeh and Yale N. Patt. 1992. A Comprehensive Instruction Fetch Mechanism for a Processor Supporting Speculative Execution. In *Proceedings of the International Symposium on Microarchitecture*. 129–139.

[64] Yi Zhang, Steve Haga, and Rajeev Barua. 2002. Execution History Guided Instruction Prefetching. In *Proceedings of the International Conference on Supercomputing*. 199–208.