PDede: Partitioned, Deduplicated, Delta Branch Target Buffer

Niranjan Soundararajan
niranjan.k.soundararajan@intel.com
Processor Architecture Research Lab, Intel Labs, India

Peter Braun
pvbraun@ucsc.edu
University of California, Santa Cruz
USA

Tanvir Ahmed Khan
takh@umich.edu
University of Michigan
USA

Baris Kasikci
barisk@umich.edu
University of Michigan
USA

Heiner Litz
hlitz@ucsc.edu
University of California, Santa Cruz
USA

Sreenivas Subramoney
sreenivas.subramoney@intel.com
Processor Architecture Research Lab, Intel Labs, India

ABSTRACT

Due to large instruction footprints, contemporary data center applications suffer from frequent frontend stalls. Despite being a significant contributor to these stalls, the Branch Target Buffer (BTB) has received less attention compared to other frontend structures such as the instruction cache. While prior works have looked at enhancing the BTB through more efficient replacement policies and prefetching policies, a thorough analysis into optimizing the BTB's storage efficiency is missing. In this work, we analyze BTB accesses for a large number (100+) of frontend bound applications to understand their branch target characteristics. This analysis, provides three significant observations about the nature of branch targets: (1) a significant number of branch instructions have the same branch target, (2) a significant number of branch targets share the same page address, and (3) a significant percentage of branch instructions and their targets are located on the same page. Furthermore, we observe that while applications' address spaces are sparsely populated, they exhibit spatial locality within and across pages. We refer to these multi-page addresses as *regions* and we show that applications traverse a significantly smaller number of regions than pages. Based on these insights, we propose *PDede*, an efficient re-design of the BTB micro-architecture that improves storage efficiency by removing redundancy among branches and their targets. *PDede* introduces three techniques, (a) BTB Partitioning, (b) Branch Target Deduplication, and (c) Delta Branch Target Encoding to reduce BTB miss induced frontend stalls. We evaluate *PDede* across 100+ applications, spanning several usage scenarios, and show that it provides an average 14.4% (up to 76%) IPC speedup by reducing BTB misses by 54.7% on average (and up to 99.8%).

CCS CONCEPTS

• **Computer systems organization → Superscalar architectures**; **Pipeline computing**.

KEYWORDS

Superscalar cores, Branch Target Buffer, Performance

ACM Reference Format:
Niranjan Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. *PDede*: Partitioned, Deduplicated, Delta Branch Target Buffer. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480046

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480046

1 INTRODUCTION

The CPU frontend bottleneck is a well-known performance problem across several usage scenarios including web-scale data center applications[6, 16, 28, 39, 50]. Due to the large code footprints of these applications, the size of the instruction working set often exceeds the microarchitectural resources of contemporary processors, such as the instruction cache (Icache), the instruction translation lookaside buffer (ITLB), the branch direction predictor, and the branch target buffer (BTB). The inability of these components to deliver instructions fast enough to the processor leads to *frontend-stalls*, significantly reducing the overall instructions per cycle (IPC) performance of a system. For instance, Google has reported that 23.5% of all CPU cycles are lost to frontend stalls for its Web search binary [8].

Prior work [5, 7, 8, 16–18, 29, 31, 34–36, 39, 42, 43, 48, 49, 53, 58] has observed the significance of the frontend bottleneck and proposed instruction prefetching techniques to address it. By predicting the control flow of applications and prefetching instructions into the Icache, a steady flow of instructions can be provided to the CPU pipeline even when the instruction working set exceeds processor resources. While instruction prefetching is effective, it only addresses one aspect of the frontend challenge, namely Icache misses, however, branch target buffer (BTB) misses significantly contribute to frontend stalls as well. In particular, we find that the limited BTB capacity leads to a significant number of BTB misses, resulting in *BTB-resteer* events and pipeline flushes. Furthermore, until the BTB miss is detected, the processor executes wrong path instructions, potentially polluting the Icache and additional structures. Figure 1 shows a Top-Down analysis [57] of over 100 frontend-bound applications, identifying BTB induced resteers are the largest contributor of frontend stalls accounting for over 40% of all frontend stall cycles. In addition to Icache misses, which has been the focus of most prior works, addressing the frontend latency bottleneck from BTB

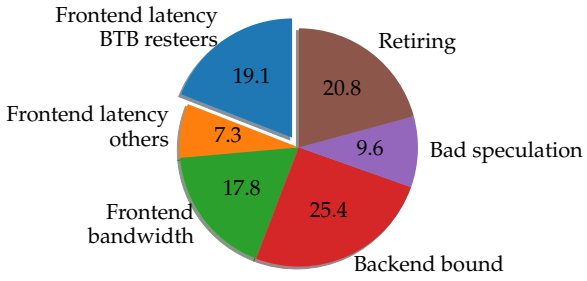


Figure 1: Frontend stall and branch resteering for 100+ applications leveraging the Top-Down methodology [57] captured on Skylake-like core.

resteers hence warrants a detailed analysis. To understand the nature of the BTB capacity problem, we perform a comprehensive study across these 100+ frontend-bound applications. We find that (1) frontend-bound applications with large code footprints, many libraries, and high branch frequencies span only 6% unique pages across all targets, in memory. (2) There exist 30% duplicate branch targets among different branch instructions, and (3) the branch PC and the branch target are located in the same page in over 60% of all cases.

Based on our findings, we study several microarchitectural techniques to address the BTB capacity problem: (1) *Branch Target Deduplication* stores a single branch target shared by multiple branches only once to improve the storage efficiency of the BTB, translating into an IPC gain of 1.6%. (2) *BTB Partitioning* breaks the BTB into separate structures, each capturing different portions of the branch target. While prior works [46] have explored partitioning, these studies lack a comprehensive analysis of branches and their targets, in particular, in the context of a modern aggressive OOO core. Our work contributes by introducing an improved partitioning technique (regions and pages) and recognizes the impact of a partitioned cache on the lookup latency by introducing and evaluating a two-cycle lookup BTB. Our combined partitioning techniques result in an additional 5.3% IPC improvement. Finally, we propose (3) *Delta Branch Target Encoding*, a technique that further increases space efficiency by optimizing branches whose PC and target reside in the same page. For these branches, we derive the page and region address from the PC and hence only need to store the page offset of the target. This results in a significant reduction in the distinct number of pages stored in the BTB, further improving storage efficiency and providing an additional 2.5% IPC gain. With Delta Branch Target Encoding, we observe that supporting both branches with targets in the same page (referred to as *same-page branches*) and in a different page (referred as *different-page branches*) leads to less efficient use of the BTB storage. We propose and evaluate two different designs that make better use of the available storage. In the first *PDede*-Multi Target design, we opportunistically support multiple same-page branches targets within a single BTB entry. This novel technique dynamically packs multiple targets in the same BTB entry without increasing the overhead for additional tags. On top of the prior gains, packing multiple targets into a single entry provides an additional 2% IPC gain. In the second *PDede*-Multi Entry size design, we support both same-page and different-page branches by providing

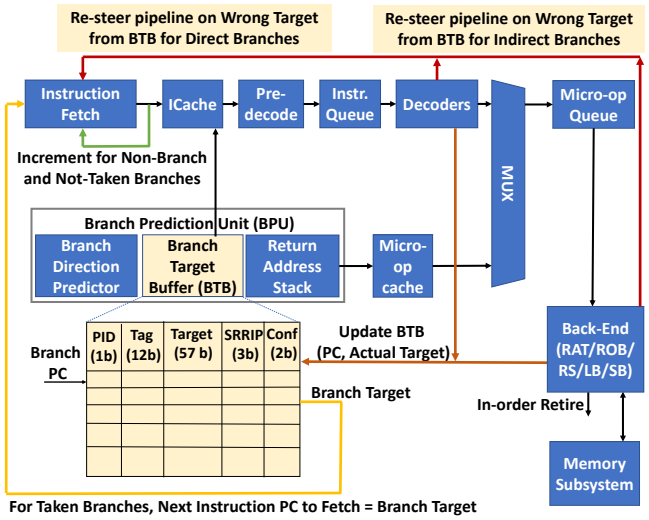


Figure 2: Pipeline describing the OOO core we study. Modern OOO cores incorporate a fetch-directed instruction prefetching (FDIP) [26] pipeline. On BTB misses/mispredictions, resteering happen post decode for direct branches or post execution for indirect branches.

variable-length entries in the BTB. This re-design of the BTB entries provides an additional 5% IPC gains on top of the delta encoding resulting mostly from re-distributing the BTB storage to support additional PCs. We implement the microarchitectural modifications proposed by *PDede* in our industry-class, in-house, cycle-accurate simulator modeling the latest Intel Icelake processor [1] and study *PDede*'s effectiveness of reducing BTB misses for a large number (100+) of frontend bound applications. *For all such applications, our best performing design reduces BTB misses significantly (54.7% on average and up to 99.8%) compared to a similar-size baseline BTB.* This results in a mean IPC speedup of 14.4% (up to 76%) across the wide range of (100+) modern CPU applications. Additionally, to provide similar mispredictions as the baseline BTB, *PDede* lowers the storage requirements by 50% highlighting the effectiveness of our changes.

2 BACKGROUND

Modern out-of-order (OOO) cores leverage superpipelining and superscalar execution to process hundreds of instructions simultaneously. To achieve high utilization of these pipelines, the processor *frontend* is required to deliver multiple instructions per cycle to the *back-end* of the core.

One of the program characteristics that limit high utilization in the pipeline are branch instructions that determine the control path in the program flow. Branches typically get classified into,

- **Conditional, Direct Branches** such as loops and forward conditionals (if-then-else) that are executed based on a condition to determine the `taken` or `not-taken` branch direction. The target address is encoded as part of the instruction.
- **Unconditional, Direct Branches** including function calls and constructs like `goto` that jump to a different address in

the code. These branches are always taken and the target address is encoded as part of the instruction.
- **Unconditional, Indirect Branches** including function calls and jumps for which the target location is unknown at compile time. These branches are also always taken.

Taken branches require the processor to fetch instructions starting from the *branch target* instead of just fetching the next sequential instruction. If the branch target is unknown, pipelines need to effectively stall, significantly reducing throughput [28, 40]. To address this challenge, processors support a branch target buffer (BTB) to predict the target address of all branch types discussed above. Figure 2 shows a typical fetch-directed instruction prefetching(FDIP) based OOO core as studied in this work. Every cycle, the frontend reads instructions from the Icache, decodes them, and forwards them to the functional units for execution. As seen in Figure 2, decoding an instruction itself happens several cycles after it enters the pipeline and, as such, branches need to be accurately predicted well in advance to guarantee a steady flow of instructions.

If the BTB cannot provide the correct branch target, for direct branches, the pipeline resteering only happens once the branch target is decoded. For indirect branches, the resteering happens only once the branch completes execution. Given that in modern OOO cores where instructions potentially spend tens to hundreds of cycles in the pipeline, BTB updates happen speculatively once the target address is known. The BTB is implemented as a cache, storing the most recent branch targets of taken branches in an application. The BTB is searched using the branch PC where a subset of the PC address bits are interpreted as the *index* selecting one set from the BTB. Each BTB entry stores the branch target as well as metadata for implementing a replacement policy (e.g., Static Re-Reference Interval Prediction (SRRIP) [27]) and maintaining counters that indicate the confidence in a target prediction. For the BTB analysis, we assume an 8-way set associative BTB with 4096 entries resembling those used in recent BTB works [3]. The branch PC used for accessing the BTB and the targets stored in the BTB are virtual addresses. Branch targets are 57 bits wide, supporting recent processors with 5-level paging [25]. BTB entries utilize a restricted 12-bit tag to disambiguate branch PCs mapping to the same cache set. Tags utilize less than $address - index$ bits to reduce storage overheads. Utilizing a smaller tag introduces the possibility of multiple branches aliasing to the same entry, forcing a *resteer* event, but it does not affect the correctness of instruction execution. With a good hashing technique, as adopted in this work, such resteering can be minimised. The per-entry confidence counters are incremented on a successful target prediction and represent a measure of the usefulness of a BTB entry, relevant for indirect branches. In particular, as the BTB stores a single target address per branch PC, it utilizes the confidence value to determine whether an indirect target address is frequently used and hence valuable or whether it should be replaced with a different target address. Finally, the SRRIP bits capture a priority order in which entries of a set can be replaced. They help implement the BTB's replacement policy, in case new branch PCs need to be inserted into the BTB. The only control-flow changing instructions that do not consume BTB entries are `returns` that are handled via the return address stack (RAS). In Section 5.6, we also evaluate alternate baseline

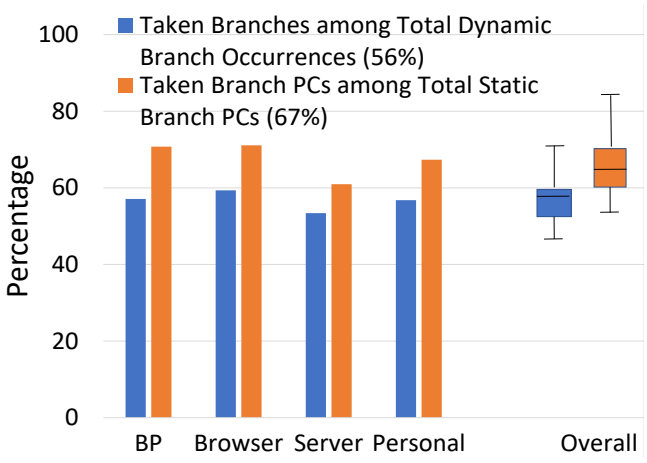


Figure 3: Percentage of static branch PCs and dynamic branch occurrences that are taken.

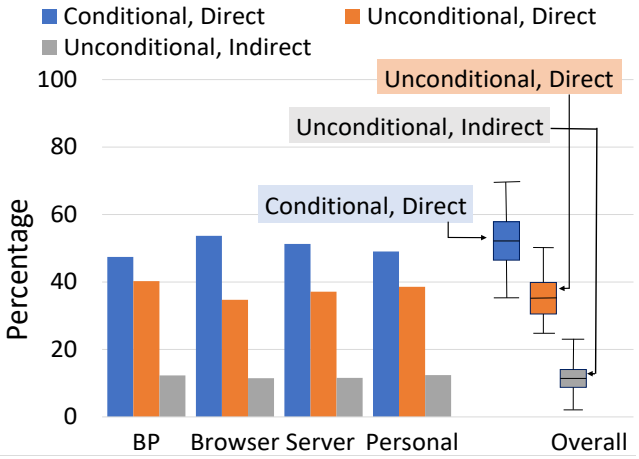


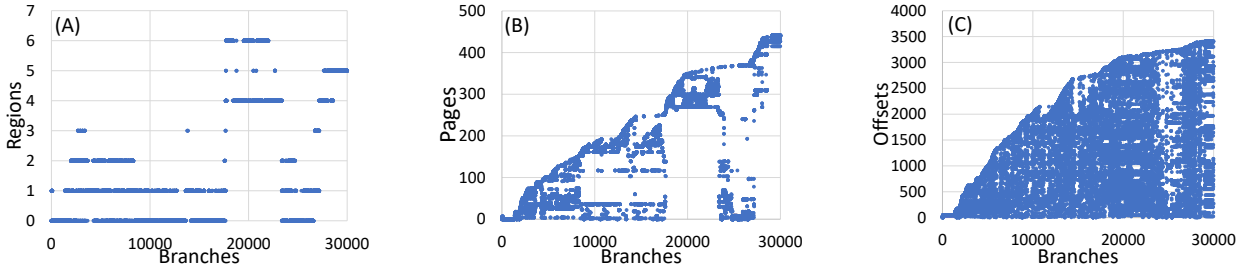
Figure 4: Percentage of the total branches belonging to the different branch types averaged per category.

BTB configurations. While most of our analysis is on a single level BTB, we do evaluate *PDede* for multi-level BTBs in section 5.9. Our observations and proposed optimizations are equally applicable for other BTB organizations as well [10].

3 BTB ANALYSIS

To guide the BTB microarchitecture we propose in the next section, we conduct a comprehensive analysis of 102 frontend-bound applications listed in Table 1. They are picked from an internal repository consisting of thousands of real world applications (including open source and proprietary ones). The selected applications are a representative set of widely used applications [1] showing a high percentage of front-end stalls (refer section 1) with branch resteering being a significant contributor.

[1]The exact listing of applications in the benchmark suite is anonymized due to the private nature of the data.



(a) Region access over time. Region differ by 65K pages in the address space and typically different libraries get dynamically mapped across these regions.

(b) Pages access over time. Less temporal locality and more unique pages accessed with respect to regions.

(c) Page offsets accessed over runtime. There is very little locality to exploit.

Figure 5: Runtime plot from a web assembly application showing the region, page and offset span of the branch targets.

Category	Description	Count
Server	Online transaction processing, Web traffic processing, Cloud services, Microservices	61
Browser	HTML5-based, Javascript, JVM, Web assembly, Games, Image-rendering	20
Business Productivity(BP)	File compression, Email, Presentations, Spreadsheet, Document processing	11
Personal	Email, Image Editing, Games, Video playback/sharing	10

Table 1: Evaluated application categories

3.1 Frequency of Taken Branches

Branch instructions only consume entries in the BTB if they are taken, as for non-taken branches the fall-through address can be computed trivially. As a result, BTB capacity is only an issue for applications whose branch working set exceeds the size of the BTB and whose branches are frequently taken. Figure 3 shows the percentage of taken branches among all static branch PCs and dynamic branch instructions. From both metrics, it can be seen that branches are taken more than 50% of the time, indicating that BTB capacity plays a critical role in determining frontend performance. Given that Icaches capture only a limited portion of the instruction footprint, for modern data center applications, a highly effective BTB that captures the targets effectively can hide a large portion of the Icache stalls by providing the target locations that need to be fetched into Icache.

Observation: Significant number of branches are taken, especially in front-end bound applications.
Insight: *Optimizing BTB storage space efficiency is required to store more branch targets and improving performance.*

3.2 Branch Type Classification

Prior work has proposed microarchitectural techniques focusing on specific branch types [15]. In this section, we show that such targeted techniques are not sufficient because datacenter applications execute a variety of branch types at runtime. We analyze three common branch types introduced in §2. Figure 4 shows a breakdown of these different branch types among all taken branches seen in the analyzed applications. While the distribution is skewed towards conditional and unconditional direct branches, all three branch types occur frequently enough and, therefore, need to be considered when designing BTBs.

Observation: BTBs need to cope with a variety of branch types including conditional, unconditional, direct, and indirect branches.
Insight: *A generic solution supporting all different branch types is required.*

3.3 Target Region and Page Partitioning

In emerging large code footprint workloads, utilizing many dynamically-mapped shared libraries, application code is spread sparsely across several pages in the address space. Operating systems also employ address space layout randomization [54] to spread out application and library code in clusters across the address space for security reasons. These address clusters could be separated by several thousand pages. Typically, different static and dynamic libraries get loaded across these different address clusters. Applications, periodically jump across these multi-page address clusters, which we refer as *regions*. As such, there is good spatial and temporal locality when applications execute in a specific region. Figure 5, shows a runtime plot across branches for a web-assembly based browser application [55]. As seen from Figure 5b, the application executes instructions from 450 different pages. Some of these pages are separated by >65K pages in the address space. However, when focusing on regions as seen in Figure 5a, addresses exhibit significant temporal and spatial locality. Our analysis, later, shows that the number of regions are fewer than the pages by 100× which we exploit for significant storage savings. Across all workloads, as shown in Figure 6, we that each page can hold about 18 branch targets while each region includes about 2200 branch targets.

This insight motivates us to split the target addresses into *regions*, *pages* and *offsets*, thereby reducing the number of unique regions and pages compared to the number of unique full addresses. When examining the offset addresses within a page, as seen in Figure 5c, we did not observe much spatial or temporal locality anymore as the address utilization within a page is dense and used equally

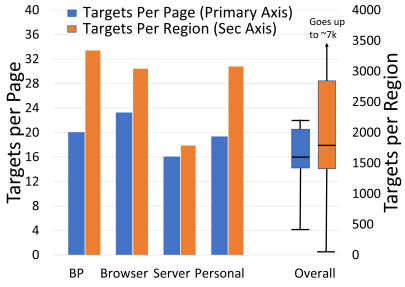


Figure 6: The average number of targets seen per page and region across the different applications.

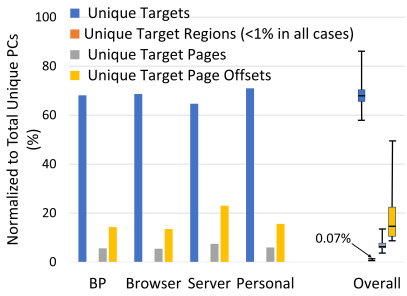


Figure 7: Unique number of branch targets, target page addresses, and target page offsets.

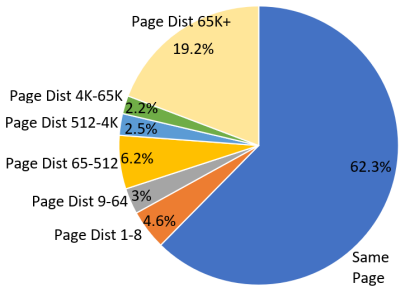


Figure 8: Distance in pages between the branch PC and its target

by different branch targets. As the three components exhibit significantly different density and spatial locality, we can adjust the number of entries that should be stored in the BTB for each of the three components individually, providing high coverage at a small storage footprint.

> **Observation:** Target addresses can be decomposed into region, page, and offset, exhibiting significantly different density and spatial locality.
> **Insight:** *Storing region, page, and offset address components separately, enables improving the storage space efficiency of the BTB. Based on this insight we propose* ***BTB-Partitioning****.*

3.4 Branch Target Address Sharing

BTBs are indexed with the PC of a branch instruction to serve the corresponding target address. There exists several high-level programming constructs such as loops with multiple conditional `breaks` or `continue` where the target is identical among several branches. Figure 7 shows the number of unique branch target addresses, branch target regions, branch target page addresses as well as branch target page offsets among all analyzed branches. The number of unique targets is 67% of the total number of unique branch PCs, indicating that 30% of the target addresses can be deduplicated. As discussed in section 3.3, the deduplication opportunities is further increased when partitioning the branch target into region, page, and page offset, yielding only 0.07%, 5% and 18% unique entities, respectively. Based on these findings, we propose *Branch Target Deduplication* in Section 4.2, a technique that stores every target region and page only once to minimize BTB occupancy. This technique eventually enables the BTB to support a larger number of branch PCs since the storage required to track the respective target addresses is significantly reduced. As shown in Section 3.3, page offsets are dense, leaving little opportunity for deduplication. As a result, *PDede* does not deduplicate page offsets and stores them explicitly per branch.

> **Observation:** Multiple branch targets share the same region and page address bits.
> **Insight:** *Deduplicating BTB entries can further improve BTB storage space-efficiency by re-using the same region and page address bits among multiple branches. Based on this insight we propose* ***Branch Target Deduplication****.*

3.5 Page Sharing between Branch and Target

The most frequently executed branch instructions are often tight inner loops. If these loops only contain a few instructions, it is likely that the branch PC and the branch target reside within the same page. We refer to these branches as *same-page branches*. On the other hand, calls frequently redirect the control flow to an instruction that is far away from the branch PC. These branches which have their target in a different page are referred to as *different-page branches*. Figure 8 analyzes the distance between the branch PC and its target for the different branch types, showing that in over 60% of the cases, the branch PC and target reside within the same page. As a result, for a given branch, the target address can be derived from its branch PC. Based on this finding, we propose *Delta Branch Target Encoding*, a new technique that only stores the page offset of the branch target in the BTB for same-page branches. To improve the BTB storage efficiency, in section 4, we propose two designs that can effectively utilize the overall BTB storage to better pack same-page and different-page branches.

> **Observation:** Branch PCs and branch targets are often located on the same page.
> **Insight:** *Based on this insight, we propose* ***Delta Branch Target Encoding*** *to only store offsets for branches whose targets are in same page.*

4 *PDEDE* ARCHITECTURE

Based on our analysis in the previous section, we discuss the microarchitectural changes required to support the three techniques to improve the space efficiency of BTBs. By increasing space efficiency, we can increase the number of branches in the BTB while keeping the overall storage capacity constant, thereby, improving the performance of frontend-bound applications. A high-level depiction of the architecture is shown in Figure 9.

4.1 BTB-Partitioning

As seen in Figure 2, the target address (57b) consumes most of the storage in the BTB. Given that there exists a significant difference in the spatial locality exhibited by regions, pages, and offsets, as

shown in Section 3.3, *PDede* exploits this insight by storing the three branch target components in different structures of differing sizes. In contrast to the baseline BTB storing 4K entries, our design utilizes 1k entries for the Page-BTB and just 4 entries for the Region-BTB and supports the 4K page offsets as is. BTB-partitioning, however, introduces a new challenge. When implemented as separate structures, the Page-BTB and Region-BTB are both required to duplicate tag and maintain other meta data for each branch PC. *PDede* eliminates the duplicate tag and meta data by introducing a level of indirection, through a structure we refer to as the *BTB-Monitor* (BTBM). More details about the BTB-Monitor are provided in the next section.

4.2 Branch Target Deduplication

Deduplicating the region and page addresses as motivated in Section 3.4 requires multiple branch PCs to refer to the same entry in the Page and Region-BTBs. *PDede* achieves this via the BTB-Monitor (BTBM), a structure that is indexed with the branch PC address and that stores pointers to the Page-BTB and a Region-BTB in each entry. The BTB-Monitor improves storage efficiency in two ways. First, the indirection (which maps a branch PC to a Page-BTB entry and an Region-BTB entry) allows multiple branch PCs to point to the same page and region entries, enabling deduplication. Second, since *PDede* maintains the tags in the BTB-Monitor, the Page-BTB and the Region-BTB do not require tags nor duplicate meta data. Page offsets are directly stored in the BTB-Monitor to avoid any separate structure or level of indirection. This is because page offsets do not exhibit locality and hence a relatively large (4K) Offset-BTB would be required. Instead of storing a 12-bit pointer into the Offset-BTB, the BTBM stores the 12-bit offset directly. We perform an analysis of the required storage resources and overheads in Section 4.4.3.

The BTBM lookup needs to be performed before accessing the Page and Region-BTBs, which introduces a lookup latency challenge. Note that, the lookup in the Page and Region-BTBs is a simple memory addressing operation that does not require an associative lookup with tag matching as in conventional caches. *Instead, tag matching is only performed for the BTBM.* We quantify the overall BTB access latency when accessing the two structures sequentially in section 5.4. Furthermore, we design and evaluate a new BTB mechanism, that can hide most of the performance penalty of the two-cycle BTB lookup. We explored an alternate *multi-tag BTB* design option in which the Page and Region-BTBs are extended to store tags so a single region or page entry can be re-used across multiple branch PCs. We opted against this option as it suffers from two disadvantages. First, multiple tags per entry increase the tag overhead, and second, the number of tags statically limits the number of branches that can have the same target.

4.3 Delta Branch Target Encoding

In Section 3.5, we showed that for over 60% of all the branches, the branch PC and the branch target reside in the same page. Our proposed delta branch target encoding scheme exploits this fact by only storing the page offset of the branch target, omitting the rest of the fields. To store only the target offset, and use the branch PC to get the rest of the target, we add an additional *delta-bit* to each entry of the BTBM identifying that the branch target is in the same page as the branch PC. When a branch PC is looked-up from the BTBM, the delta bit determines whether a) the branch target is formed by concatenating the Region-BTB entry, Page-BTB entry and offset, or b) the branch target is formed by concatenating information from the branch PC with the offset from the BTBM. This technique significantly reduces the number of distinct pages and regions that need to be tracked in the BTBM. Furthermore, since the offset information is available in the BTBM entry, it avoids the need to lookup Region/Page-BTB lookup, eliminating the one additional cycle latency for same-page branches. As we show later in Figure 11b , opportunistically eliminating the extra cycle access improves the performance gains that *PDede* can provide.

4.3.1 Optimizations enabled by Delta-Encoded BTB. Omitting the page and region addresses for same-page branches reduces the number of required entries in the Page- and Region-BTB, however, it still wastes storage capacity by storing null pointers in the BTBM. In the following, we explore two different techniques to eliminate this storage overhead as well. In the first optimization, referred to as *PDede*-Multi Target and shown in Figure 9B, we opportunistically re-use the Region- and Page-BTB pointer fields in the BTBM to store targets of other branches, under the following conditions.

- The two stored branches need to be same-page branches
- The branch PC of the second branch represents the next taken branch after first branch in the instruction sequence. In this case, the next taken branch follows the target of the first branch.

To retrieve a target from the *PDede*-Multi Target we perform the following operations. Every time an entry is read out from the BTBM, we check for a valid next target and store it in a temporary global 12-bit *Next Target Offset* register. To enable this, every BTBM entry is extended with a valid bit (Next Target bit). If the next taken PC misses in the BTBM, we use the offset from Next Target Offset register to provide the target for this (next taken) PC. Note that, the target might not always be correct but since the PC missed in the BTBM and will resteer eventually, providing the target from Next Target Offset register does not increase the resteers. This technique has two advantages. Firstly, the technique optimizes for same-page branches which, according to section 3.5, represent 60% of all branches. Secondly, it is only invoked for the immediate taken branch, following the PC, that misses in the BTBM. As such, there is limited additional storage required per entry (only 1 bit per entry). Figure 9 highlights these changes. There are several extensions to this idea, in terms of adding simple tags to provide targets beyond the next taken branch or having multiple Last BTBM set and way registers to improve BTBM utilization, which we plan to explore later.

We explore a second design alternative, referred to as *PDede*-Multi Entry size, that restructures the BTB to support two different entry sizes. In each set of the BTBM, half of the ways are reserved for same-page branches and avoid the region and page pointer fields. Only the other half include these page and region pointers. We redistribute the storage savings to increase the total number of BTBM entries. Such a static split of ways might not be optimal

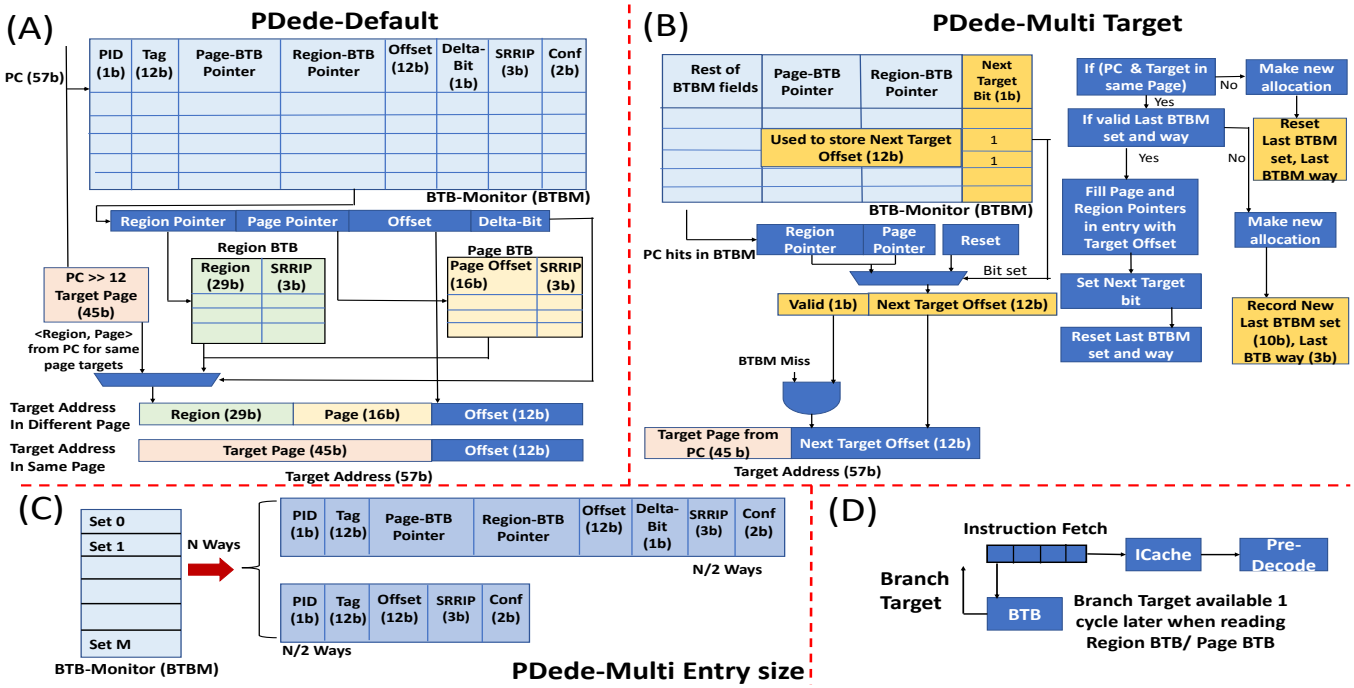


Figure 9: (A) *PDede* BTB Architecture with Region- and Page-BTBs emphasizing how the tables provide different portions of the branch target. (B) *PDede*-Multi Target design which supports multiple targets in an entry when branches that have their targets in the same page. Changes required are shown in yellow. (C) *PDede*-Multi Entry size design to support multiple entry sizes in different ways of the same set in the BTBM. Branches with targets in different pages cannot allocate in entries that do not have Region and Page Pointers. (D) *PDede* introduces a 1 cycle stall to provide branch targets for different-page branches.

HW Structure	Element size (bits)	Entry size	Entries	Size
Baseline BTB	Target address(57), Tag(12), SRRIP(3), Confidence(2), PID(1)	75 bits	4096, 8-way	37.5KB
BTBM-Default	tag(12), Page-Pointer(10), Region-Pointer(2), Offset(12), SRRIP(3) confidence(2), PID(1), delta(1)	43 bits	6144, 6-way	32.3 KB
BTBM-Multi-target	tag(12), Page-Pointer(10), Region-Pointer(2), Offset(12), SRRIP(3) confidence(2), PID(1), delta(1), Next target (1)	44 bits	6144, 6-way	33 KB
BTBM-Multi-entry size	Support 2 entry sizes: Default & Default-(Region, Page Pointers, delta bit)	43 bits/30 bits	8192, 8-way	36.5 KB
Region-BTB	Region(29), SRRIP(2)	31 bits	4, 4-way	0.02KB
Page-BTB	Page offset(16), SRRIP(4)	20 bits	1024, 16-way	2.5KB
PDede-Default	BTBM-Default, Region-BTB, Page-BTB			34.8KB
PDede-Multi Target	BTBM-Multi target, Region-BTB, Page-BTB, Additional registers			35.5KB
PDede-Multi Entry size	BTBM-Multi entry, Region-BTB, Page-BTB			39KB

Table 2: Storage Requirements of *PDede* and the Baseline BTB

across all applications but, as shown in Table 2, even if the application only has targets in different pages, the number of *suitable* entries match the baseline BTB and, therefore, at least provide the baseline performance. Figure 9C shows the two different entry types supported.

4.4 Putting it all Together

Figure 9A shows the overall microarchitecture for *PDede*. The BTBM serves the incoming branch PC lookups and stores pointers to the Region-BTB and Page-BTB. The BTBM maintains a 12-bit tag per entry to disambiguate branch PCs with the same index. Each BTBM entry also contains a 1 bit process ID (PID), 2 bits for SRRIP-based replacement, 1 delta-bit and 2 bits for prediction confidence.

4.4.1 Lookup Operation. To perform a BTB lookup, the index bits from the branch PC address are used to read entries from the corresponding set from the BTBM. Next, the tag bits from the branch PC are compared to the BTBM entries of the set and, if there is a match, the Region and Page-BTB pointers as well as the offset and the delta-bits are retrieved. If the delta bit is set, the branch PC is

concatenated with the target offset, immediately returning the target without incurring the additional 1 cycle penalty. Furthermore, in the case of *PDede*-Multi Target, if the Next Target Bit is set as well, the region and page pointers are reinterpreted and stored into the Next Target Offset register. If the next BTBM lookup misses, the target for that PC will be served from the Next Target Offset register, otherwise the register is cleared. No additional tasks need to be performed for the *PDede*-Multi Entry size implementation. If, on a BTB lookup, the delta bit is not set, the page and region pointers are used to lookup the corresponding entries from the Page-BTB and Region-BTB. The region, page, and offset address bits are concatenated to form the target address. In this case, due to the two sequential lookups in the BTBM and the Page-BTB, the total BTB lookup latency is higher than the baseline BTB. In this work, we conservatively assume an extra cycle to complete the read access (since the cycle time is really decided by the most critical path in the pipeline). Once the region pointer is know, the Region-BTB access happens in parallel with the Page-BTB access.

4.4.2 Update/Allocation Operations. Once a branch has been decoded or executed, the correct target information is known. The confidence counters in the BTBM are updated based on the correctness of the final target prediction.Note that for each update operation, meta information (confidence counter, replacement bits) in the BTBM, Region-BTB, and Page-BTB need to be updated. One difference between *PDede* and the baseline BTB is that the allocation in the BTBM is only made if the allocations in the Region and Page-BTBs have been successful, avoiding invalid entries in the BTBM. Allocations in the Region-BTB and Page-BTB are SRRIP-guided [27]. SRRIP is sufficient as an entry shared across multiple often-occurring targets will always remain in the corresponding table. These tables are indexed using the corresponding portion from the target address (region and page) so as to detect if a particular region or page already exists in the table and allocation is done only when needed. We do not do anything special to invalidate the pointers in the BTB Monitor when entries in Page-BTB or Region-BTB get replaced. The reason is that if the Region-BTB or Page-BTB entry was popular it would not be replaced as it would continuously be accessed by one of the entries pointing to it. But it is possible for a BTBM entry to read an updated Page BTB entry leading to the wrong target. Such cases were very rare (0.06%) and hence we did not find the need to add the additional complexity to clear the BTBM entries when page and region BTB entries get replaced.

In the *PDede*-Multi Target design, if the branch that is allocated has its target in the same page, then its BTBM set and way are recorded in the *Last BTBM set* and *Last BTBM way* registers. When the next taken branch, whose target is in the same page, is updated, then using the Last BTBM set and way registers, the BTBM entry is updated to hold a valid next target offset and next target bit is also set. If the next taken branch is not a same-page branch, the Last BTBM set and way registers are cleared. In the *PDede*-Multi Entry size design, the only change is that allocations are restricted for different-page branches as only half the ways, per set, are available.

4.4.3 Storage Requirements. Table 2 provides an architecturally feasible *PDede* configuration whose size matches the baseline BTB as close as possible to enable an ISO-storage comparison.

5 EVALUATION

We now evaluate the impact of *PDede* on BTB mispredictions per kilo instructions (MPKI) and instructions per cycle (IPC) performance via simulation. We break down the IPC performance gains across the three proposed techniques and provide sensitivity studies to evaluate different BTB configurations and storage sizes. We also study the latency impact of accessing the two BTB structures sequentially.Finally, we also provide an iso-MPKI configuration and show the significant storage reduction that *PDede* can enable.

5.1 Methodology

Our performance evaluation is done using our in-house cycle-accurate execution driven simulator modeling an x86 core clocked at 3.9 GHz. The simulator models pipeline latency based on functional unit contention, resource back-pressure due to data dependencies, a decoupled frontend, resteering due to branch misprediction and accurate wrong path modeling. The microarchitectural parameters are similar to the latest Intel Icelake processor [1] and we list the relevant parameters in Table 3.

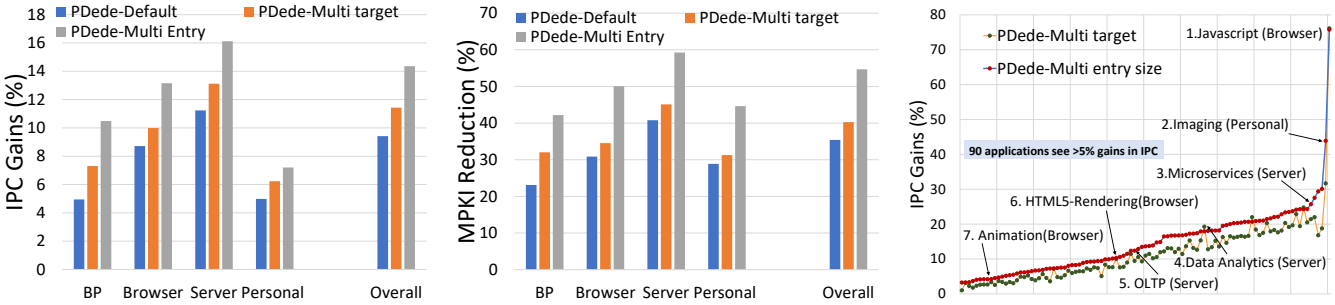
As the baseline BTB, we leverage the architecture described in Section 2. We define BTB misses as follows. A BTB miss occurs if either a) a branch PC does not have a valid entry in the BTB, of b) if a branch PC is contained in the BTB, however, the target address is incorrect. To evaluate, *PDede* we use the applications introduced in Section 3. We leverage Simpoints [23] to identify the regions of interest and run detailed simulations with 10M+ instructions after warming up the memory sub-system and microarchitectural structures using 100M+ instructions. While the MPKI reduction and IPC gains presented below are normalized with the baseline, none of the applications had low baseline MPKI as their performance are bound by the BTB re-steering (refer Figure 1). Our simulator and application performance estimation methodology is correlated to within 5% error to real silicon in the market.

Core	6-wide OOO, 352-entry ROB, 32-entry fetch queue, 72-entry load buffer, 128-entry store buffer
Branch Predictor	CBP-2016 [11] - 64 KB
Baseline/*PDede* BTB	Refer Table 2
Uop cache	2.25K entries
L1 cache	Private, 48KB, 64B line, 8 way, prefetchers on
L2 cache	Private, 512KB, 64B line, 8 way, prefetchers on
LLC	Shared, Inclusive, 2MB, 64B line, 16 way
Main Memory	Dual channel DDR4-3200MHz

Table 3: Simulator Parameters

5.2 IPC and MPKI Performance of *PDede*

Figure 10a, 10b shows the performance benefits of *PDede* over the baseline configuration, shown in Section 4.4.3, for three configurations of *PDede*. Overall, *PDede*-Default improves IPC by 9.4% by lowering the BTB MPKI by 35.4% across the 100+ applications. The Server category containing large code footprint webscale applications shows the highest reduction in MPKI of 40.7% resulting in IPC gains of 11.2%. The IPC gains are a direct result of reducing the front-end stalls provided by the increased effective BTB capacity



(a) IPC gains across the different categories of applications

(b) BTB MPKI reduction across the different categories

(c) Per-application IPC gains curve highlighting some of the applications. All applications gain more than 3%.

Figure 10: IPC and MPKI improvements provided by *PDede*

that reduces branch re-steering events. The overall MPKI reduction is a direct result of *PDede*'s ability to track more branch PCs in the BTBM. A 50% larger baseline BTB provides the same IPC gains except for requiring an additional 37KB. *PDede*-Multi Target improves upon *PDede*-Default with its ability to opportunistically store 2 targets in a single BTBM entry. The IPC gains increase to 11.4% due to the 5% additional MPKI reduction provided by supporting more targets in the same storage footprint. The *PDede*-Multi Entry size, on the other hand, increases the IPC gains to 14.4% by storing targets for twice the number of branches as baseline. All branch types experience a decrease in MPKI including Indirect branches, which have a much higher misprediction penalty, that show a 4% decrease in mispredictions. Conditional direct branches see 74% reduction while unconditional direct branches see 49% reduction in MPKI.

Figure 10c shows the IPC gains for all of the 100+ evaluated applications comparing the *PDede*-Multi Target and *PDede*-Multi Entry size configurations. For clarity, we omit the *PDede*-Default configuration. As can be seen, IPC gains range from 3% to 76%. Figure 10c highlights specific applications such as Javascript static analyzer, which benefits the most, seeing a 76% IPC gain resulting from a 99.8% reduction in BTB MPKI. This BTB MPKI reduction lowers the branch re-steering frontend stalls seen in this application by 75% translating directly into IPC gains. Similar performance improvements can be observed in the Imaging and Microservice applications showing >18% IPC gains. On the other hand, in the Animation application, the BTB MPKI is lowered only by 23% since the hot code working set of the application is large, exceeding *PDede*'s resources. The Animation application has a 2.3× larger page footprint than the Javascript static analyzer and, hence, only sees a limited IPC gain for existing BTB sizes.

PDede is well-equipped to handle the different requirements of the diverse applications we analyze in this paper. For instance, in the Data Analytics application, which has a high percentage of branches whose targets occur in the same page as the branch PC (90%), the BTBM provides the targets. Not only does it benefit from limited front-end stalls but we also observe that this workload gains more with *PDede*-Multi Target as multiple targets in the same page can be packed better in this design than *PDede*-Multi Entry size. In the Microservices and the OLTP applications, on the other hand, the branch PCs and their corresponding targets frequently span different pages (only 50% of branch PC and targets are in same page in these applications), yet the Region and Page-BTBs combine effectively to capture the targets. Even in *PDede*-Multi Entry size, where only half the entries can support targets across different pages, there are enough entries available to support the varying dynamic requirements across the different branch types to result in the MPKI reduction. The HTML5-rendering application highlights the benefit of region and page deduplication. This application exhibits an average of > 15 branch targets for each page and $> 2K$ per region maximizing the efficiency of the Page and Region BTBs.

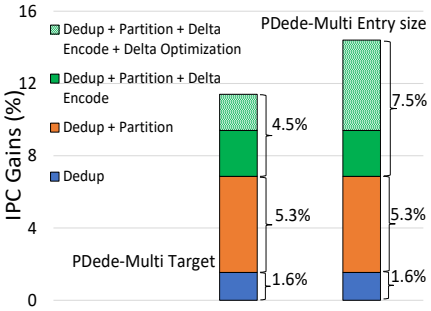
Going forward, for the next sections, we use *PDede*-Multi Entry size to study the sensitivity to the different parameters.

5.3 Sensitivity to design observations

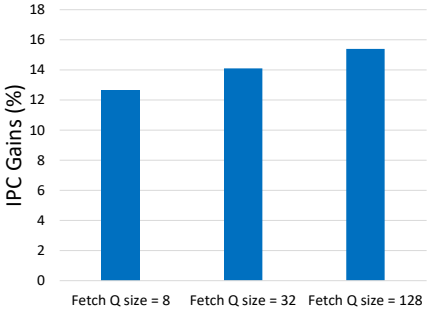
To quantify the impact of the main observations on which *PDede* is architected on, Figure 11a shows the IPC gains obtained by each optimization technique. Deduplicating targets (which are 67% of branch PCs) alone is not very effective and provides only 1.6% IPC gains. Once the targets are split into regions and pages, and individually deduplicated, the gains are much more significant (5.3%). This is in line with what was observed in Figure 7. Finally, adding delta branch encoding to the two techniques increases IPC by an additional 2.5%. Further, the multi-target and multi-entry size optimizations, enabled by our delta branch encoding, improve the IPC gains significantly by a further 2% and 5% respectively.

Component	**Access Time (ns) 1 RW port**	**Access Time (ns) 6 RW ports**
Baseline BTB	0.24	0.72
BTBM	0.21	0.55
Page-BTB(PBTB)	0.09	0.16
PDede (BTBM+PBTB)	0.3	0.71

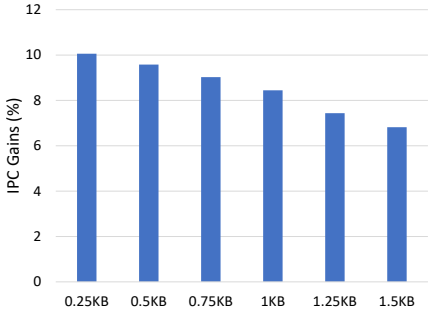
Table 4: Access Latency comparison at 22nm. For *PDede*, access latency shown for default cacti parameters.



(a) Contributions from the different observations that together helped architect *PDede*



(b) IPC gains when BTB stalls for every taken branch and across different fetch queue sizes



(c) IPC gains in a 2 level BTB organization with the Level1 BTB rearchitected based on *PDede*. The gains are shown for different sizes of Level0 BTB.

Figure 11: *PDede* IPC contributions and sensitivity in different configurations

5.4 *PDede* access latency

Using Cacti7 [56], we studied the access latency of the baseline BTB and *PDede* at 22nm in Table 4. The critical path for *PDede* access is the BTBM access followed by an access to the Page-BTB. 22nm is the most recent technology supported by Cacti7. We highlight two different configurations, one utilizing 1 read-write (RW) port and another with 6 RW ports. Note that even a 1-RW design can support multiple BTB accesses per cycle when leveraging a banked design while the 6 port design matches the pipeline width. As seen, *PDede* has higher access latency than the baseline BTB only when sequentially accessing the Page-BTB. Otherwise, the access latency of the BTBM is lesser than the baseline BTB's access latency. Hence, *PDede*, only incurs a 1 cycle penalty to provide the branch target when accessing the page and region BTBs is required. We also evaluated the impact when the BTB access for every taken branch take two cycles, irrespective of whether it requires an access to the Region-BTB and Page-BTB or not. This configuration lowers the overall IPC gains from 14.4% to 13.4%. In a decoupled frontend [26, 44], delaying the BTB lookup by one cycle is not exposed to later pipeline stages. The additional cycle renders it more challenging for the frontend to run-ahead, however, since typical basic blocks involve a few instructions, the branch predictor can proceed quickly enough to predict future branches. Figure 11b shows the IPC impact when varying the fetch queue size. As expected, the gains are lower at smaller fetch queue size (12.7%) which increases as the fetch queue becomes larger (15.4% when fetch queue has 128 entries). In a FDIP pipeline, the fetch queue controls how far ahead the branch predictor and BTB (at high accuracy) can fetch and hence the IPC gains scale accordingly.

5.5 *PDede* performance with Perfect Branch Direction Predictor

To analyze the interactions between the branch direction predictor and the BTB, we studied the effectiveness of *PDede* with a perfect branch direction predictor improving the overall IPC gains over the baseline from 14.4% to 15.2%.

5.6 Impact of Indirect Branches

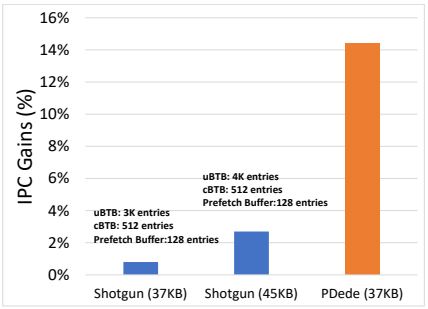
Modern cores incorporate an Indirect Target TAGE (ITTAGE) [45] predictor to predict the target of indirect branches. The storage requirements of ITTAGE (64KB) is quite high, although indirect branches only represent 10% of total dynamic branches. We evaluate *PDede* and the baseline BTB system with an additional 64-KB ITTAGE. For both designs, indirect branch targets are not allocated in the BTB. *PDede* provides a 13.9% IPC gain over the baseline BTB. IPC improvements are lower than in the configuration without ITTAGE, as the indirect branch MPKI reduction provided by *PDede* does not apply in this case. Also, the baseline BTB now has additional entries available for storing direct branches.

5.7 Impact of Storing Return Instructions

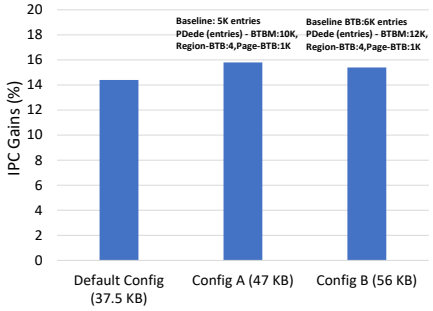
The RAS is common in most modern architectures as it can store the targets of return instructions with perfect accuracy. Nevertheless, to reduce storage costs and, for simplicity, architectures may opt to omit the RAS and instead store return targets in the BTB. We see a 13.7% IPC gain for *PDede* over the baseline BTB when storing return targets in the BTB. Note that our sizing analysis performed in Section 3 does not include return targets and hence Page and Offset-BTBs may be sized non-optimally for this configuration.

5.8 Sensitivity at larger BTB Sizes

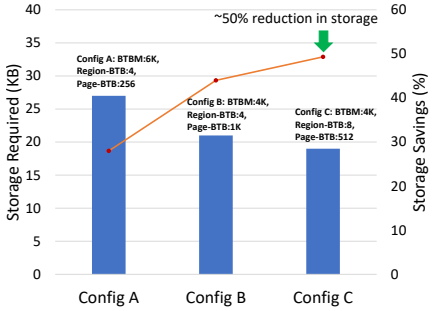
In Figure 12b, we highlight the IPC gains provided by *PDede* at larger BTB sizes. These gains primarily stem from *PDede*'s ability to track more PCs in the BTBM (6K additional PCs tracked in ConfigB compared to baseline) and both the Region and Page-BTBs not having to scale up much to support the additional targets. We continue scaling the BTB size to 16K entries (150KB), and PDede still provides a 3.3% IPC improvement across 100+ applications at iso-storage. The IPC gains are lower as the active footprint of several applications start to fit in this size. Nevertheless, for the JITed server applications with large footprints, IPC gains of 6% are still significant. To be iso-MPKI, with the 150KB baseline BTB, PDede only requires 87 KB. This results in 42% lower storage resulting in area and energy savings, and as workload footprint sizes are continuing to increase [28], PDede will continue to provide significant benefits.



(a) IPC comparison with state-of-art BTB design from Shotgun [35]



(b) *PDede* IPC gains as the BTB storage budget is scaled.



(c) Iso-MPKI *PDede* configurations

Figure 12: *PDede* IPC sensitivity at larger sizes, in future cores and storage savings possible at Iso-MPKI

5.9 Sensitivity for 2-level BTB designs

In Figure 11c, we study *PDede* in a 2-level BTB configuration [22]. We study a baseline configuration with Level0 BTB at multiple sizes providing predictions at 1 cycle latency along with a 4K-entry Level1 BTB providing predictions at 2 cycles. In this setup, *PDede* is used to optimize only the Level1 BTB and yet it provides significant IPC gains. Further, as the Level1 BTB size grows, as shown in Figure 12b, *PDede* will continue to remain efficient and provide significant IPC benefits over the conventional BTB organization.

5.10 Comparison to state-of-art BTBs

In Figure 12a, we show that the state of art BTB design, Shotgun [35], brings only about 2.7% IPC speedup over the baseline BTB studied in this work (4K-entries at 37.5KB). As seen, Shotgun gives only about 0.8% IPC gains over the baseline, at iso-storage, which increases to about 2.7% at 45KB. Note that we did not model the Return Instruction Buffer (RIB) and instead use the RSB to provide target address for returns similar to baseline. Prefetching the conditionals following the return was modelled similar to what was done in [35]. Several factors contribute to Shotgun's lower gains. These include the need to capture targets of all taken/not taken conditional branches in CBTB, which lower its hit rate. The baseline BTB is PC-indexed and therefore only needs to capture taken branches in it. Further, the prefetching is only triggered when a prior unconditional branch hits in the uBTB and only the conditional branches within a certain offset from the unconditional branch are prefetched. Prefetching into ICache, on top of FDIP[26], also pollutes the ICache entries due to the high speculation in the front-end.

Confluence [29] and SN4L [5] operate at the cache line granularity and store the branch information in the cache line as meta-data. For RISC-ISAs, given the fixed instruction length, simple bit vectors suffice to capture the meta-data. For the x64 CISC ISA, there is a need to store branch offsets which is much harder to accommodate. SN4L estimates that it would require at least an additional 65KB metadata, virtualized, for a 2K-entry BTB. For Confluence, the meta-data that needs to be stored is much larger (960KB). This is discussed in Section V-D of SN4L paper. The high storage costs make these techniques harder to adopt. On the other hand, PDede provides all its gains at iso-storage.

PDede can definitely complement Confluence, Shotgun, and other BTB prefetching techniques to hold more branches in the BTB and in turn reduce the prefetching needed.

5.11 *PDede* with Deeper Future Pipelines

Modern OOO cores continue to increase in their pipeline width and depth to extract more single-thread performance. A side-effect of this growth is the performance penalty of BTB mispredictions increases as the pipeline stages between when the BTB prediction is given and when the actual branch target is available increases. Therefore, lowering the BTB MPKI will only become more critical in future cores. To study this impact, we scale the pipeline parameters listed in Table 3 by 1.5x and 2x of their current size to reflect future cores. The IPC gains from *PDede* BTB increases to 16.8% in the 1.5x Icelake-like core which increases further to 20.1% in the 2x Icelake-like core.

5.12 Iso-MPKI *PDede* Storage Savings

While the prior sections studied the performance impact of iso-storage *PDede* configuration in different scenarios, storage savings are equally important both from cost and energy savings perspective. As such, in Figure 12c, we also studied the smallest size *PDede* BTB that is iso-MPKI with the baseline BTB. The smallest *PDede* configuration requires only 19KB, resulting in 49% storage savings over the baseline.

6 RELATED WORK

Significant effort has gone in to studying the frontend bottleneck and proposing techniques to lower its impact [5, 7, 8, 17, 18, 29–32, 39, 42]. By predicting the control flow of applications and prefetching instructions into the Icache and BTB, a steady flow of instructions can be provided. *PDede* is complementary to these works. Recent papers from Samsung and IBM [2, 22] have shown the significant investments made in BTB storage emphasizing the importance of lowering BTB resteers. Also, as shown in Figure 12b, *PDede* still gives good performance gains at larger BTB sizes.

Seznec [46] proposed the page pointer table, which has also been used by Garza [19], to improve the storage efficiency of the BTB via partitioning. While the proposed technique is related, *PDede* improves over the page pointer table by splitting the page address into

regions and page offsets and deduplicating the individual structures to help lower the storage costs. *PDede* introduces Delta Encoding of branch targets to further improve storage efficiency. Delta Encoding enables *PDede*-Multi Target to opportunistically store multiple targets in the same entry which has not been explored till now. Further, it allows the entries to be sized differently, bringing in significant IPC benefits via the *PDede*-Multi Entry size design. Lastly, for a thorough evaluation, *PDede* takes the performance implication of accessing multiple structures into account. Our analysis and newly introduced techniques improve the performance by an additional 7.5% over implementations that only support partitioning (Figure 11a).

Several works have looked to enhance the efficiency of the BTB by applying different replacement policies. These works [9, 41] investigate different BTB allocation strategies for different branch types. GHRP [3] was introduced to improve the BTB replacement policy which is again orthogonal to our work and can be combined with *PDede*. Several proposals [15, 20, 33, 37, 38, 51] aim to reduce misspredictions for indirect branches. *PDede* contributes over these works by tackling all branch types while showing a MPKI reduction for indirect branches as well. The Phantom-BTB [10] increases effective BTB capacity by adding a virtual second level BTB in the L2 cache, prefetching branch metadata into the BTB to mitigate the added latency. This solution still leverages a dedicated BTB component whose storage efficiency can be improved with *PDede*.

Several prior studies on caches have exploited data redundancy to lower the storage requirements either by deduplicating [12–14, 24, 47, 52] the data or by proposing efficient compression [4, 21]. Such compression techniques are infeasible for BTBs which need to provide fast lookups whereas *PDede* has been optimized for performance.

7 CONCLUSION

In this paper we present *PDede*, the Partitioned, Deduplicated, Delta Branch Target Buffer. *PDede* proposes three novel techniques to increase the space-efficiency of BTBs leading to significant performance gains for frontend-bound applications. Our Delta-branch target encoding is novel and allows us to store multiple targets in a single entry and we evaluate the impact of supporting different entry sizes. In an iso-storage configuration, *PDede* shows an average BTB MPKI reduction of 54.7% and an average IPC improvement of 14.4% over the baseline BTB. Alternately, *PDede* achieves iso-MPKI at 49% lower storage than baseline. We believe that our analysis provides insights that eases future research on this topic to develop better branch target buffers increasing the efficiency and throughput of contemporary microprocessors.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback and suggestions. This work was supported by the Intel Corporation, the NSF grants #1823559 and #2010810, and the Applications Driving Architectures (ADA) Research Center, a JUMP Center cosponsored by SRC and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 2019. Icelake. https://www.anandtech.com/show/14514/examining-intels-ice-lake-microarchitecture-and-sunny-cove/.
[2] N. Adiga, J. Bonanno, A. Collura, M. Heizmann, B. R. Prasky, and A. Saporito. 2020. The IBM z15 High Frequency Mainframe Branch Predictor Industrial Product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 27–39.
[3] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A Jiménez. 2018. Exploring predictive replacement policies for instruction cache and branch target buffer. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 519–532.
[4] A. R. Alameldeen and D. A. Wood. 2004. Adaptive cache compression for high-performance processors. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.* 212–223.
[5] Ali Ansari, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2020. Divide and Conquer Frontend Bottleneck. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*.
[6] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory hierarchy for web search. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 643–656.
[7] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 513–526.
[8] 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 46th International Symposium on Computer Architecture*. 462–473.
[9] Brian K. Bray and M. J. Flynn. 1991. Strategies for branch target buffers. In *Proceedings of the 24th annual international symposium on Microarchitecture - MICRO 24*. ACM Press, Albuquerque, New Mexico, Puerto Rico, 42–50. https://doi.org/10.1145/123465.123473
[10] Ioana Burcea and Andreas Moshovos. 2009. Phantom-BTB: a virtualized branch target buffer design. *Acm Sigplan Notices* 44, 3 (2009), 313–324.
[11] CBP-5. 2016. Championship Branch Prediction (CBP-5). https://www.jilp.org/cbp2016/.
[12] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. 2014. CMD: Classification-based memory deduplication through page access characteristics. *ACM SIGPLAN Notices* 49, 7 (2014), 65–76.
[13] David Cheriton, Amin Firoozshahian, Alex Solomatnikov, John P. Stevenson, and Omid Azizi. 2012. HICAMP: architectural support for efficient concurrency-safe shared structured data access. *ACM SIGPLAN Notices* 47, 4 (March 2012), 287–300. https://doi.org/10.1145/2248487.2151007
[14] Timothy E Denehy and Windsor W Hsu. 2003. Duplicate Management for Reference Data. *Research Report RJ10305, IBM* (2003), 15.
[15] M. Farooq, L. Chen, and L. Kurian. 2010. Value Based BTB Indexing for indirect jump prediction. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/HPCA.2010.5416659
[16] 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. *Acm sigplan notices* 47, 4 (2012), 37–48.
[17] Michael Ferdman, Cansu Kaynak, and Babak Falsafi. 2011. Proactive instruction fetch. In *International Symposium on Microarchitecture*.
[18] Michael Ferdman, Thomas F Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. 2008. Temporal instruction fetch streaming. In *International Symposium on Microarchitecture*.
[19] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez. 2019. Bit-level Perceptron Prediction for Indirect Branches. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 27–38.
[20] E. Garza, S. Mirbagher-Ajorpaz, T. A. Khan, and D. A. Jiménez. 2019. Bit-level Perceptron Prediction for Indirect Branches. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 27–38.
[21] Amin Ghasemazar, Prashant Nair, and Mieszko Lis. 2020. Thesaurus: Efficient Cache Compression via Dynamic Clustering. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 527–540.
[22] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 40–51.
[23] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. SimPoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction Level Parallelism* (2005), 1–28.

[24] Bo Hong and Demyn Plantenberg. 2004. Duplicate Data Elimination in a SAN File System. *MSST 2004* (2004), 14.

[25] Intel. 2017. *5-Level Paging and 5-Level EPT*. Technical Report. Intel.

[26] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo. 5555. Rebasing Instruction Prefetching: An Industry Perspective. *IEEE Computer Architecture Letters* 01 (oct 5555), 1–1. https://doi.org/10.1109/LCA.2020.3035068

[27] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 60–71. https://doi.org/10.1145/1815961.1815971

[28] 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 42nd Annual International Symposium on Computer Architecture*. 158–169.

[29] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: unified instruction supply for scale-out servers. In *Proceedings of the 48th International Symposium on Microarchitecture*. 166–177.

[30] 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 *54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[31] Tanvir Ahmed Khan, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2020. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 146–159.

[32] Tanvir Ahmed Khan, Dexin Zhang, Akshitha Sriraman, Joseph Devietti, Gilles Pokam, Heiner Litz, and Baris Kasikci. 2021. Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA 2021)*.

[33] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn. 2009. Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware. *IEEE Trans. Comput.* 58, 9 (2009), 1153–1170.

[34] Aasheesh Kolli, Ali Saidi, and Thomas F Wenisch. 2013. RDIP: return-address-stack directed instruction prefetching. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 260–271.

[35] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the front-end bottleneck with shotgun. *ACM SIGPLAN Notices* 53, 2 (2018), 30–42.

[36] Rakesh Kumar, Cheng-Chieh Huang, Boris Grot, and Vijay Nagarajan. 2017. Boomerang: A metadata-free architecture for control flow delivery. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 493–504.

[37] Tao Li, Ravi Bhargava, and Lizy Kurian John. 2002. Rehashable BTB: an adaptive branch target buffer to improve the target predictability of Java code. In *International Conference on High-Performance Computing*. Springer, 597–608.

[38] Tao Li, Ravi Bhargava, and Lizy Kurian John. 2005. Adapting branch-target buffer to improve the target predictability of java code. *ACM Transactions on Architecture and Code Optimization (TACO)* 2, 2 (2005), 109–130.

[39] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. 2012. Scale-out processors. *ACM SIGARCH Computer Architecture News* 40, 3 (June 2012), 500–511. https://doi.org/10.1145/2366231.2337217

[40] A. Perais, R. Sheikh, L. Yen, M. McIlvaine, and R. D. Clancy. 2019. Elastic Instruction Fetching. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 478–490. https://doi.org/10.1109/HPCA.2019.00059

[41] Chris H Perleberg and Alan Jay Smith. 1993. Branch target buffer design and optimization. *IEEE transactions on computers* 42, 4 (1993), 396–412.

[42] A. Ramirez, O.J. Santana, J.L. Larriba-Pey, and M. Valero. 2002. Fetching instruction streams. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.* 371–382. https://doi.org/10.1109/MICRO.2002.1176264 ISSN: 1072-4451.

[43] Glenn Reinman, Brad Calder, and Todd Austin. 1999. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 16–27.

[44] Glenn Reinman, Brad Calder, and Todd Austin. 2001. Optimizations Enabled by a Decoupled Front-End Architecture. *IEEE Trans. Comput.* 50, 4 (April 2001), 338–355. https://doi.org/10.1109/12.919279

[45] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor. In *JILP*. https://hal.inria.fr/hal-00639041

[46] S. Seznec. 1996. Don't Use the Page Number, but a Pointer to It. In *23rd Annual International Symposium on Computer Architecture*. IEEE Computer Society, Los Alamitos, CA, USA, 104. https://doi.org/10.1145/232973.232985

[47] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. 2017. Pageforge: a near-memory content-aware page-merging architecture. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 302–314. https://doi.org/10.1145/3123939.3124540

[48] Lawrence Spracklen, Yuan Chou, and Santosh G Abraham. 2005. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In *International Symposium on High-Performance Computer Architecture*.

[49] Viji Srinivasan, Edward S Davidson, Gary S Tyson, Mark J Charney, and Thomas R Puzak. 2001. Branch history guided instruction prefetching. In *International Symposium on High-Performance Computer Architecture*.

[50] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. Softsku: Optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*. 513–526.

[51] D-CD Tang, Ann Marie Grizzaffi Maynard, and Lizy Kurian John. 1999. Contrasting branch characteristics and branch predictor performance of C++ and C programs. In *1999 IEEE International Performance, Computing and Communications Conference (Cat. No. 99CH36305)*. IEEE, 275–283.

[52] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. 2014. Last-Level Cache Deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing* (Munich, Germany) *(ICS '14)*. Association for Computing Machinery, New York, NY, USA, 53–62. https://doi.org/10.1145/2597652.2597655

[53] Alexander V. Veidenbaum. 1997. Instruction cache prefetching using multilevel branch prediction. In *High Performance Computing (Lecture Notes in Computer Science)*, Constantine Polychronopoulos, Kazuki Joe, Keijiro Araki, and Makoto Amamiya (Eds.). Springer, Berlin, Heidelberg, 51–70. https://doi.org/10.1007/BFb0024203

[54] Wikipedia contributors. 2018. ASLR. https://en.wikipedia.org/wiki/Address_space_layout_randomization.

[55] Wikipedia contributors. 2018. WebAssembly. https://en.wikipedia.org/wiki/WebAssembly.

[56] S. J. E. Wilton and N. P. Jouppi. 1996. CACTI: an enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits* 31, 5 (1996), 677–688. https://doi.org/10.1109/4.509850

[57] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–44.

[58] Yi Zhang, Steve Haga, and Rajeev Barua. 2002. Execution history guided instruction prefetching. In *Proceedings of the 16th international conference on Supercomputing (ICS '02)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/514191.514220