



Article

BTIP: Branch Triggered Instruction Prefetcher Ensuring Timeliness

Wenhai Lin ¹, Yiquan Lin ¹, Yiquan Chen ², Shishun Cai ², Zhen Jin ¹, Jiexiong Xu ¹, Yuzhong Zhang ² and Wenzhi Chen ^{1,*}

¹ College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China; linwh@zju.edu.cn (W.L.); linyiquan@zju.edu.cn (Y.L.); jin_zhen@zju.edu.cn (Z.J.); jasonxu@zju.edu.cn (J.X.)

² Alibaba Group, Hangzhou 311121, China; zy.zhengyi@alibaba-inc.com (Y.C.); shishun.css@alibaba-inc.com (S.C.); yuzhong.zy@alibaba-inc.com (Y.Z.)

* Correspondence: chenwz@zju.edu.cn

Abstract: In CPU microarchitecture, caches store frequently accessed instructions and data by exploiting their locality, reducing memory access latency and improving application performance. However, contemporary applications with large code footprints often experience frequent Icache misses, which significantly degrade performance. Although Fetch-Directed Instruction Prefetching (FDIP) has been widely adopted in commercial processors to reduce Icache misses, our analysis reveals that FDIP still suffers from Icache misses caused by branch mispredictions and late prefetch, leaving considerable opportunity for performance optimization. Priority-Directed Instruction Prefetching (PDIP) has been proposed to reduce Icache misses caused by branch mispredictions in FDIP. However, it neglects Icache misses due to late prefetch and suffers from high storage overhead. In this paper, we proposed a branch-triggered instruction prefetcher (BTIP), which aims to prefetch Icache lines that FDIP cannot efficiently handle, including the Icache misses due to branch misprediction and late prefetch. We also introduce a novel Branch Target Buffer (BTB) organization, BTIP BTB, which stores prefetch metadata and reuses information from existing BTB entries, effectively reducing storage overhead. We implemented BTIP on the ChampSim simulator and evaluated BTIP in detail using traces from the 1st Instruction Prefetching Championship (IPC-1). Our evaluation shows that BTIP outperforms both FDIP and PDIP. Specifically, BTIP reduces Icache misses by 38.0% and improves performance by 5.1% compared to FDIP. Additionally, BTIP outperforms PDIP by 1.6% while using only 41.9% of the storage space required by PDIP.

Keywords: computer architecture; CPU microarchitecture; CPU front-end optimization; instruction cache miss; hardware prefetching; instruction prefetching



Citation: Lin, W.; Lin, Y.; Chen, Y.; Cai, S.; Jin, Z.; Xu, J.; Zhang, Y.; Chen, W. BTIP: Branch Triggered Instruction Prefetcher Ensuring Timeliness. *Electronics* **2024**, *13*, 4323. <https://doi.org/10.3390/electronics13214323>

Academic Editors: Yongseok Son and Sunggon Kim

Received: 13 September 2024

Revised: 26 October 2024

Accepted: 1 November 2024

Published: 4 November 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Contemporary applications are becoming increasingly complex, incorporating deep software stacks such as functional logic, language runtimes [1], development frameworks [2], RPC frameworks [3], logging systems [4], and more. As a result, these applications often have large code footprints, sometimes reaching hundreds of megabytes [5–10], which far exceed the capacity of the instruction cache (Icache) or even the L2 cache. This oversized and continuously growing code footprint [8] leads to frequent Icache misses.

The frequent Icache misses degrade application performance because these misses stall the pipeline and limit the rate at which the CPU front-end delivers instructions to the CPU back-end [8,11]. Even though increasing the size of the Icache can reduce misses, it comes at the cost of additional area and power overhead. Moreover, larger Icache sizes lead to increased cache latency. Since the Icache lies on the critical path, this additional latency further degrades the overall application performance.

Instruction prefetching [12–22] is an effective technique to reduce Icache misses by predicting future instruction accesses and loading them into the Icache in advance. A next-line prefetcher [12] works well for covering contiguous Icache accesses, but its effectiveness drops significantly when faced with non-contiguous accesses caused by various branches. The temporal prefetcher can effectively solve the problem of discontinuous instruction cache access of the next-line prefetcher by tracking the time-dependent flow of instruction cache misses. Specifically, if the instruction that triggered the stream is accessed again, the subsequent address will be prefetched. However, due to the large code footprint of contemporary applications, implementing these solutions requires complex designs and impractical hardware storage overhead, which has hindered their widespread adoption in mainstream processors.

Unlike previous solutions, Fetch-Directed Instruction Prefetching (FDIP) [14,15] is an advancing instruction prefetcher that achieves both low overhead and high performance, making it widely adopted in modern processors [23–27]. FDIP adopts a front-end design that decouples the Branch Prediction Unit (BPU) from the Instruction Fetch Unit (IFU). In FDIP, instruction addresses predicted by the BPU are stored in a dedicated Fetch Target Queue (FTQ), and the prefetch engine prefetches the instructions indicated by the FTQ entries before they are fetched by the IFU. FDIP effectively reuses existing branch prediction components, including the direction predictor and the Branch Target Buffer (BTB). These highly accurate branch prediction components in modern processors enable FDIP to precisely predict instruction access with minimal hardware overhead, significantly reducing Icache misses and improving overall performance.

However, FDIP still has significant optimization opportunities for many applications. We evaluated the performance of the FDIP and found the following: (1) For the 10 most front-end intensive applications, FDIP still has an average of 14.7% performance optimization potential. (2) Branch misprediction and late prefetch are two factors that cause the Icache misses in FDIP. Specifically, 47.52% of the Icache misses are attributable to branch mispredictions, while 52.48% are caused by late prefetch. To further optimize FDIP, Priority-Directed Instruction Prefetching (PDIP) [28] was proposed, introducing a dedicated table to track Icache misses caused by branch mispredictions and trigger prefetch requests when the same branch is encountered. Although PDIP effectively reduces the number of Icache misses in FDIP due to branch mispredictions, it neglects Icache misses caused by a late prefetch. In addition, PDIP suffers from unnecessary high storage overhead due to storing too much duplicated information already stored in the BPU.

In this paper, we propose BTIP, which is a branch-triggered instruction prefetcher. BTIP aims to prefetch Icache lines that cannot be effectively prefetched by FDIP, such as the recovery path of mispredicted branches and the Icache lines with long latency. The key idea behind BTIP is to identify an appropriate branch (src-branch) to trigger the prefetching of Icache lines (dst-cachelines) likely to be missed due to branch misprediction or late prefetch. We categorize the causes of Icache misses into **branch misprediction** and **late prefetch** based on the execution intervals between dst-cachelines and src-branch. For branch mispredictions, BTIP selects the last mispredicted branch before the dst-cacheline as the src-branch. For late prefetch, BTIP selects a branch executed early enough as the src-branch by the lookup branch history queue, ensuring timely prefetching. The dst-cachelines and their corresponding src-branch are associated together and stored for future prefetching. Once the src-branch is encountered, BTIP prefetches the recorded dst-cachelines, thus mitigating Icache misses that FDIP cannot cover and improving application performance. In addition, BTIP expands some Branch Target Buffer (BTB) entries by adding fields to store prefetch metadata, allowing the reuse of existing BTB information without introducing excessive and unnecessary storage overhead.

We implemented BTIP on the Champsim simulator and evaluated BTIP in detail using traces from the 1st Instruction Prefetching Championship (IPC-1) [29]. The IPC-1 traces include 8 client application traces, 35 server application traces, 3 Spec-gcc traces, 2 Spec-gobmk traces, 1 Spec-perlbench trace, and 1 Spec-x264 trace. Our evaluation shows that

BTIP outperforms both FDIP and PDIP. Specifically, BTIP reduces Icache misses by 38.0% and improves performance by 5.1% compared to FDIP. Additionally, BTIP outperforms PDIP by 1.6% while using only 41.9% of the storage space required by PDIP.

In summary, our contributions are as follows:

- We conducted a detailed performance analysis of FDIP and found that it still has significant potential for optimization with branch misprediction and late prefetching being the two causes of Icache misses in FDIP.
- We propose a branch-triggered instruction prefetcher, BTIP, which is designed to optimize the Icache misses caused by both branch mispredictions and late prefetch, thereby enhancing application performance.
- We introduce a novel BTB organization, BTIP BTB, which stores prefetch metadata and reuses information from existing BTB entries, effectively reducing storage overhead.
- We performed a comprehensive evaluation of BTIP using 10 front-end intensive applications. The results show that BTIP reduces Icache misses by 38.0% and improves performance by 5.1% compared to FDIP. Additionally, BTIP outperforms PDIP by 1.6% while using only 41.9% of the storage space required by PDIP.

The rest of this article is organized as follows. Section 2 introduces the background relevant to this work. Section 3 evaluates the performance of FDIP and outlines the motivation behind BTIP. Section 4 describes the design of BTIP. Section 5 presents the performance results of BTIP compared to previous methods and discusses the findings. Section 6 lists the cost of BTIP and compares BTIP with FDIP and PDIP in terms of design, performance, power consumption, and storage (area) overhead. Section 7 reviews related work on instruction prefetching techniques. Finally, Section 8 concludes the article.

2. Background

2.1. Branch Prediction Unit

The Branch Prediction Unit (BPU) predicts the program's control flow and provides the Instruction Fetch Unit (IFU) with the instruction addresses (PCs) along the predicted control path. The IFU retrieves these predicted instructions and forwards them to the processor's decoding unit. Since the control flow of a program is determined by the outcomes of branching instructions, recognizing branch instructions is essential for accurate control flow prediction. However, whether a given PC corresponds to a branch instruction and its branch target address can only be determined after the instruction has been fetched and decoded. To eliminate the delays caused by fetch and decode operations, the BPU uses a hardware structure called the Branch Target Buffer (BTB), which identifies whether an instruction is a branch and determines the branch target address based on the instruction's address before it is fetched.

Figure 1 illustrates the structure of a BTB entry. Each entry consists of the fields `Valid`, `PC_Tag`, `Branch_Type`, `Target`, and `LRU`. The `PC_Tag` contains a hash of the significant bits of the PC to reduce storage overhead. The `Target` field holds the destination address of the branch with its length typically determined by the virtual address space and instruction set architecture. We take the ARMv8 ISA with a 48-bit virtual address space as an example. In ARMv8, instructions are always 4-byte aligned, meaning the last two bits of the instruction address are always 0. Therefore, only 46 bits are needed to store the branch target address. The `Valid` field indicates whether the entry is valid, while the `LRU` field stores replacement information to determine which entry to evict.

| | | | | |
|--------------|----------------|--------------------|----------------|------------|
| Valid: 1-bit | PC_Tag: 12-bit | Branch_Type: 2-bit | Target: 46-bit | LRU: 3-bit |
|--------------|----------------|--------------------|----------------|------------|

Figure 1. Composition of an entry in conventional BTB.

The BPU relies on several components working together to predict branch targets based on the input PCs, as illustrated in Figure 2. First, the BTB and Indirect Predictor are accessed simultaneously. If the PC hits the Indirect Predictor, the BPU directly outputs the branch target predicted by the Indirect Predictor. Otherwise, the BPU determines the

branch target based on the results of BTB, conditional predictor, and Return Address Stack (RAS). Like most cache structures, BTB typically employs a set-associative structure. The lower bits of the PC are used to index the BTB and locate the set, while the hash of the higher bits of the PC is compared against the PC_Tag of all entries within the set. If no matching entry is found, it indicates that the PC does not correspond to a branch instruction, and the program is executed sequentially. In this case, the BPU outputs the address of the next instruction (PC+4) as the branch target. On the contrary, if a matching PC_Tag is found, it indicates that the PC represents a branch instruction. The BPU then determines the branch target based on the branch type of the corresponding entry.

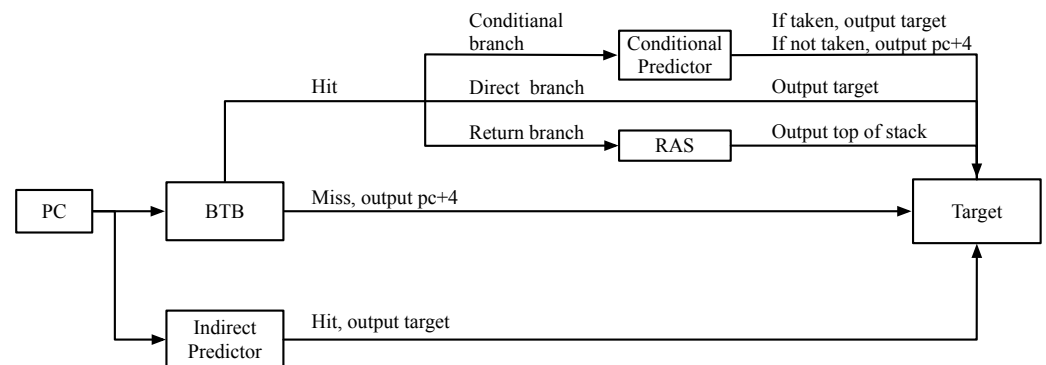


Figure 2. The process of BTB predicting a branch target.

Specifically, if the branch is direct, the BPU outputs the address stored in the branch target field. For conditional branches, the output is determined by the result of the direction predictor. If the direction predictor predicts the branch as taken, the BPU outputs the branch target address. On the contrary, if the prediction is not taken, the BPU outputs the address of the next instruction (PC+4) as the branch target. In the case of a return branch, the BPU uses the address at the top of the Return Address Stack (RAS) as the branch target instead of the BTB's target field. This is because the return address of a function depends on the address of the instruction that called the function. Each time a function call occurs, the return address is determined as the address of the calling instruction +4. The BPU pushes this return address into the RAS when a function call occurs. When a return instruction is encountered, the BPU pops the top address from the RAS and uses it as the branch target.

2.2. Fetch-Directed Instruction Prefetching (FDIP)

Figure 3 illustrates the architecture of Fetch-Directed Instruction Prefetching (FDIP). FDIP is a decoupled front-end microarchitecture. FDIP adopts a front-end design that decouples the BPU from the IFU. Unlike a traditional coupled front-end, FDIP separates BPU and IFU using a Fetch Target Queue (FTQ). The BPU consists of the BTB, a conditional branch predictor (TAGE [30]), an indirect branch predictor (ITTAGE [31]), and a Return Address Stack (RAS). The BPU is responsible for speculatively predicting the addresses of subsequent instruction blocks to be executed ①. The FTQ is a FIFO queue, which stores information about the addresses of the instruction blocks to be executed as predicted by the BPU. The prefetch engine in FDIP initiates prefetch requests to load the relevant cache lines of instruction blocks into the Icache as soon as an entry is added to the FTQ ②. If a request misses in the Icache, the prefetch engine will fetch the missing cache line from the lower levels of the memory hierarchy ③.

The IFU fetches instructions by retrieving the address of the next instruction block from the head of the FTQ ④ and loading the corresponding instruction from the Icache ⑤. Once the fetched instruction is returned, the IFU sends it to the decoder for decoding and then forwards it to the back-end for execution.

The BPU typically operates at a higher throughput than the IFU and back-end, which often results in a full FTQ. Therefore, modern processors are typically equipped with a

longer FTQ, increasing the prefetch lead time and allowing FDIP to cover both L1 and even L2 cache misses.

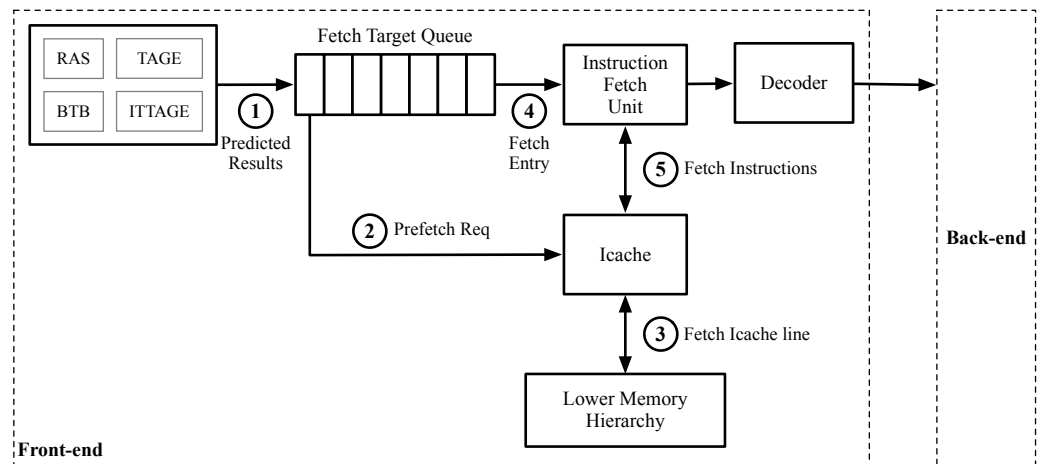


Figure 3. Overview of decoupled front-end microarchitecture design.

FDIP achieves a low-overhead, high-performance front end by leveraging the existing advanced BPU. However, its performance is limited by the prediction accuracy of the BPU, and Icache misses with too long latency. First, branch mispredictions will cause the FTQ to be flushed, leaving FDIP unable to initiate prefetches early enough to cover the latency of Icache misses. Second, too-long miss latency may exceed the maximum coverage time provided by the FTQ, resulting in prefetches that fail to fully cover the miss latency.

3. Motivation

In this section, we analyze the performance of the applications from the First Instruction Prefetch Championship (IPC-1) [29] to answer the following questions:

- Can the Icache misses in FDIP be optimized, and what is the potential optimization opportunity for FDIP? (Section 3.1);
- What are the critical factors of Icache misses in FDIP? (Section 3.2);
- Does the existing PDIP solution effectively resolve the Icache miss issue in FDIP, and does it introduce any new problem? (Section 3.3).

3.1. The Optimization Opportunity of FDIP

In this subsection, we evaluate the Icache misses per kilo instructions (MPKI) across various applications with FDIP enabled and further explore the potential optimization opportunity for FDIP.

To figure out whether FDIP completely eliminates front-end stalls, we measured the Icache MPKI across a range of applications with FDIP enabled. As depicted in Figure 4, the result indicates that FDIP does not entirely eliminate Icache misses. Specifically, several applications, such as from `server_003` to `server_013`, exhibit substantial Icache MPKI, with values close to or exceeding 8. The high MPKI suggests that these applications still encounter significant front-end stall bottlenecks and can be optimized.

To further quantify the performance optimization potential of FDIP, we measured the Instructions per Cycle (IPC) speedup using an ideal Icache (with no Icache misses) and a double-sized Icache. The results in Figure 5 reveal that on average, the ideal Icache achieves a 4.7% speedup, whereas the double-sized Icache results in only a 0.4% speedup. Moreover, for the ten most front-end intensive applications, the ideal Icache provides an average speedup of 14.8%, while the double-sized Icache offers a mere 0.2% improvement. These findings suggest that compared to an ideal Icache, FDIP has substantial potential for enhancement. Furthermore, simply increasing the Icache size yields negligible performance benefits.

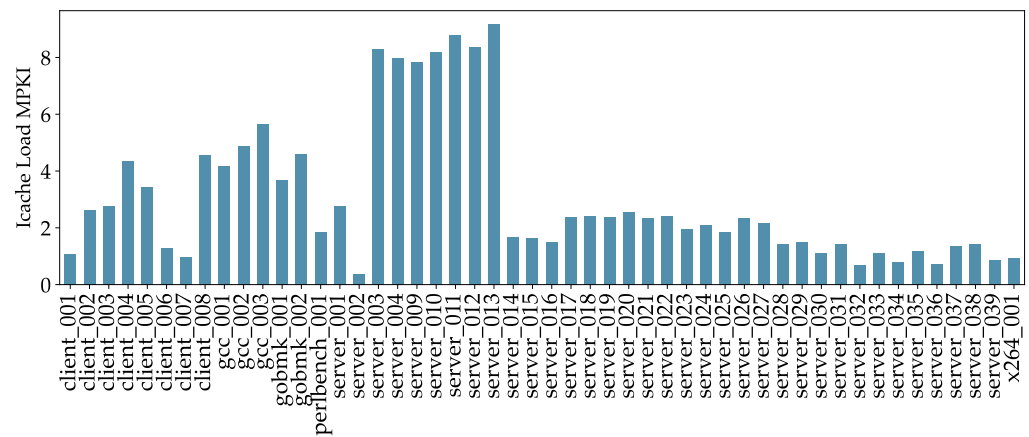


Figure 4. Icache load miss per kilo instructions with FDIP enabled.

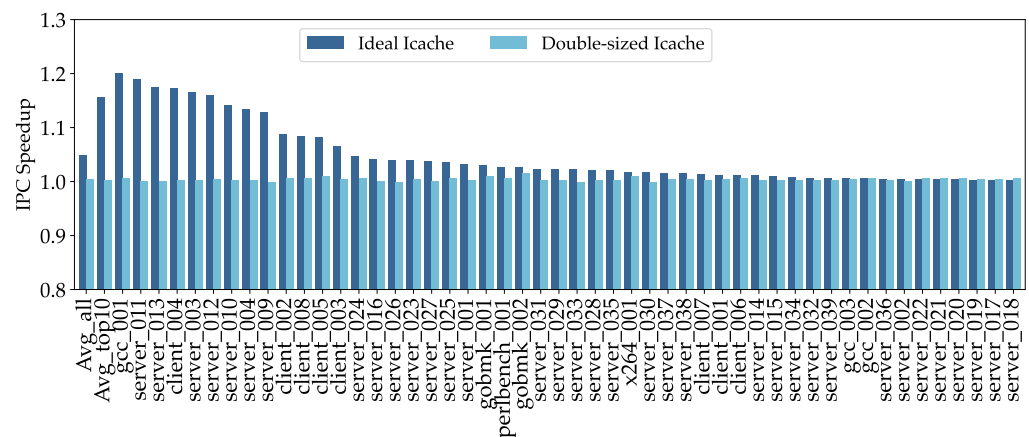


Figure 5. The IPC speedup of ideal Icache with FDIP enabled and double-sized Icache. Applications are sorted by the speedup value of the ideal Icache. Avg_all represents the averaged speedup across all applications, while Avg_top10 indicates the averaged speedup for the top 10 applications with the highest speedup using the ideal Icache.

3.2. The Causes of Icache Miss

There are two critical factors that cause the above Icache misses: *branch misprediction* and *late prefetch*. Specifically, branch misprediction would flush the FTQ in FDIP as shown in Figure 3, which results in the FDIP not having enough time to prefetch following instructions, causing Icache misses and degrading the FDIP’s performance. Additionally, even with correct branch prediction, certain instructions may take an excessive amount of cycles to be prefetched into the Icache. In this case, such instructions are fetched by the IFU (5) before being loaded into the Icache from the lower level memory hierarchy (3): for instance, when the Icache request hits in DRAM, and the long latency of DRAM hinders FDIP’s ability to prefetch instructions effectively, leading to Icache misses.

To quantitatively analyze the proportion of Icache misses caused by the above two factors, we measured the number of Icache misses under a perfect BPU scenario without branch misprediction. In the case of a perfect BPU, Icache misses caused by branch misprediction can be completely avoided, and Icache misses are only associated with late prefetch. Thus, we can obtain the proportion of Icache misses caused by icache misprediction and late prefetch, respectively. As shown in Figure 6, 47.52% of the Icache misses are attributable to branch mispredictions, while 52.48% are caused by late prefetch. Therefore, this motivates us to optimize both factors to reduce Icache miss and enhance performance.

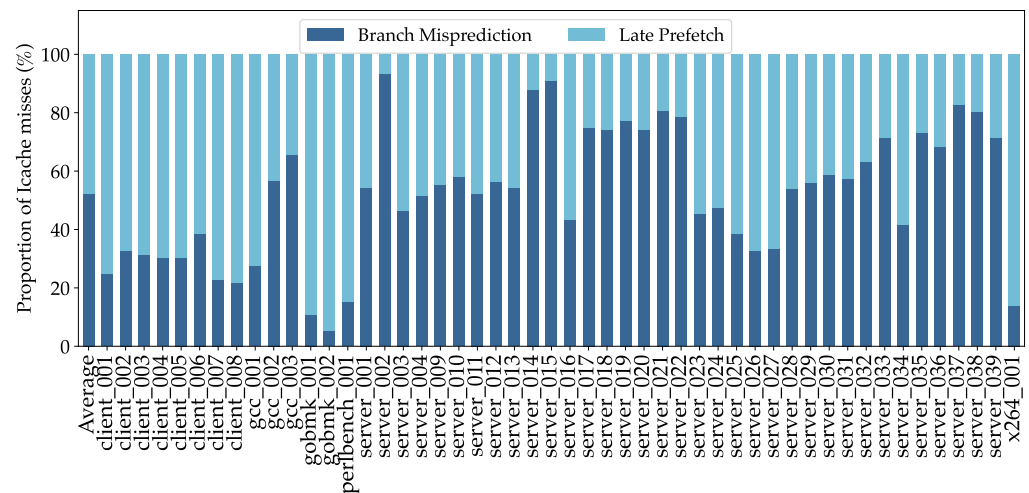


Figure 6. The proportion of Icache misses caused by branch mispredictions and late prefetches.

3.3. Disadvantages of PDIP

PDIP [28] is proposed to optimize the Icache misses caused by branch mispredictions in FDIP. In particular, PDIP uses a dedicated table (PDIP table) to record the relationship between mispredicted branches and the addresses of the missed Icache lines they cause. When BPU generates a new predicted result according to a PC, PDIP checks whether the PC has a corresponding entry recorded in the PDIP table. If an entry exists, the Icache lines recorded in that entry are prefetched. PDIP enhances Icache miss coverage and improves application performance.

However, PDIP addresses only the Icache misses resulting from branch mispredictions and does not account for those caused by late prefetch. As illustrated in Figure 6, the proportion of Icache misses attributable to late prefetch surpasses that due to branch mispredictions. Consequently, by focusing exclusively on mitigating Icache misses caused by branch mispredictions, PDIP neglects a substantial number of potential optimization opportunities.

Furthermore, PDIP also suffers from high storage overhead due to the duplication of information, such as PC_Tag and Target, between the PDIP table and the BTB. This redundancy leads to unnecessary metadata storage overhead. The rapidly increasing code footprint [6,8,32] significantly exacerbates storage overhead. We measured the probability that the PC_Tag in a PDIP table entry is also present in the BTB. The results in Figure 7 show that the average duplication probability across applications is as high as 88%. This high level of redundancy motivates us to propose a novel BTB organization to reuse the existing BTB metadata, thus significantly reducing storage overhead.

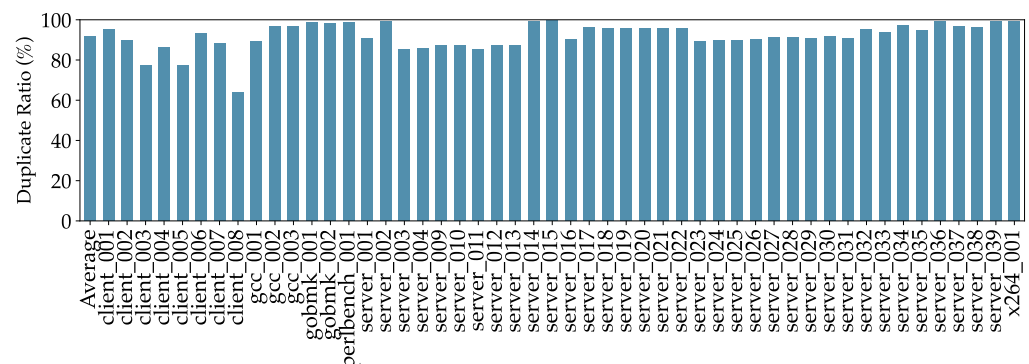


Figure 7. The probability that metadata in the PDIP table are also present in the BTB.

3.4. Summary

In this section, we conducted a series of experiments and found the following:

- FDIP does not entirely eliminate Icache misses across various applications, and there are still significant optimization opportunities for FDIP.
- Branch misprediction and late prefetch are two critical factors that cause the Icache misses in FDIP.
- PDIP focuses on Icache misses caused by branch mispredictions in FDIP, but it neglects Icache misses due to late prefetch and introduces significant storage overhead.

These findings motivate us to propose a solution that optimizes Icache misses caused by both branch mispredictions and late prefetches, enhancing FDIP performance. Additionally, the solution should reuse existing BTB information to minimize storage overhead.

4. Design

In this section, we describe the design of BTIP in detail. BTIP aims to prefetch Icache lines that cannot be effectively prefetched by FDIP, such as the recovery path of mispredicted branches and the Icache lines with long latency, as described in Section 3.

4.1. Overview

Figure 8 shows the overall microarchitecture of BTIP. BTIP introduces a new prediction unit to predict Icache lines that may be missed due to branch misprediction and late prefetch. Subsequently, BTIP sends prefetch requests for these Icache lines to the prefetch queue (PQ) to reduce Icache misses that FDIP cannot cover. The key idea behind BTIP is to identify an appropriate branch (*src-branch*) to trigger the prefetching of Icache lines (*dst-cachelines*) likely to be missed due to branch misprediction or late prefetching. Specifically, for branch mispredictions, BTIP selects the last mispredicted branch before the *dst-cachelines* as the *src-branch*. For late prefetch, BTIP selects a branch executed early enough as the *src-branch* by lookup branch history queue, ensuring timely prefetching. Subsequently, once the same *src-branch* is encountered, BTIP prefetches the recorded *dst-cachelines*, thus mitigating Icache misses that FDIP cannot cover and enhancing application performance.

As shown in Figure 8, BTIP introduces and modifies several components to help identify the appropriate *src-branch* and record the relationship between the trigger and the *dst-cachelines*. These components include the following: (1) TAGE Accuracy Table (TAT): TAT tracks the branch prediction accuracy for each bank of TAGE and ITTAGE over a time interval and outputs the hit bank at each branch prediction. BTIP can estimate the accuracy of the current prediction based on the accuracy of the bank hit by the branch prediction, filtering out prefetches triggered by branches with high prediction accuracy. (2) BTIP Table (BT): It is responsible for recording prefetch metadata of indirect branches and direct branches. Since BTB does not store entries for indirect branches, we need to dedicate a table for indirect branches to store prefetching metadata. (3) Branch History Queue (BHQ): BHQ records the history of branches and the execution time of each branch. It is used to find suitable branches as *src-branches* for late prefetch Icache misses. (4) Prediction Unit: It predicts Icache lines that may be missed due to branch misprediction and late prefetch and then sends prefetch requests for these Icache lines to the PQ. (5) Modified MSHR: We extend MSHR to record the timestamp when an Icache miss occurs in order to calculate its latency.

In addition, BTIP extends existing BTB entries to store prefetch-related metadata, thus reducing storage overhead. Specifically, the BTB is divided into two tables: the Conventional BTB and the BTIP BTB. The structure of the Conventional BTB remains unchanged, while the entry of the BTIP BTB is extended to store prefetch-related metadata, particularly the *src-branch* and *dst-cachelines* pairs. By reusing fields in the BTB rather than creating a dedicated table for storing prefetch metadata, BTIP achieves lower storage overhead compared to PDIP.

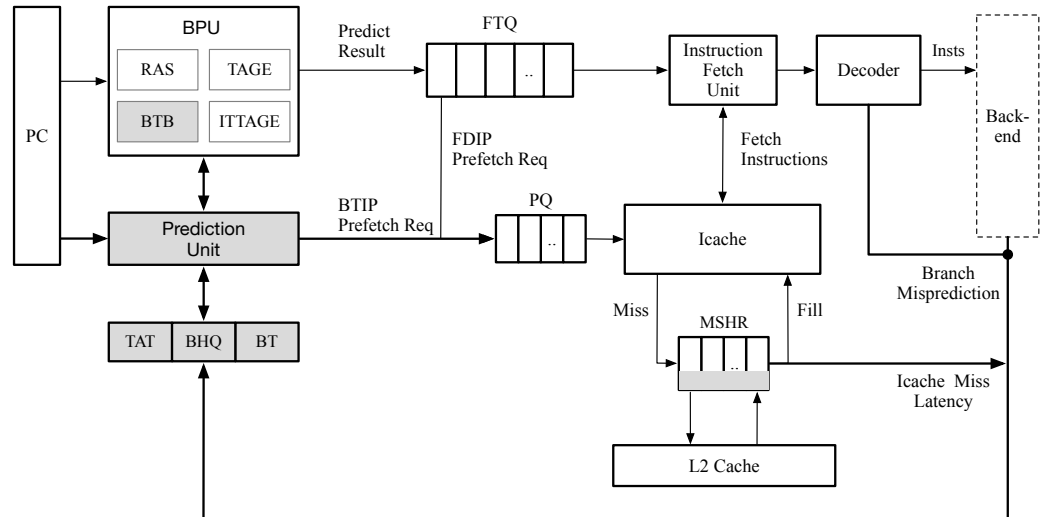


Figure 8. The overall microarchitecture of BTIP, where the gray blocks are newly added components or modified components.

4.2. Building Prefetch Metadata

As described in the previous section, we divide the causes of Icache miss into two categories: **branch misprediction** and **late prefetch**. Figure 9 provides an example to illustrate how to build prefetch metadata for triggering prefetching. First, we identify the dst-cachelines that were missed due to the reasons mentioned above. Next, we determine the appropriate src-branch for these dst-cachelines. Finally, the dst-cachelines and their corresponding src-branch are linked together and stored in the BTIP BTB or BTIP Table for future prefetching.

Branch Misprediction. BTIP assigns each instruction an ID number to represent its execution order. When an instruction (such as *A* in Figure 9) experiences an Icache miss, ① BTIP calculates ΔID , defined as $ID_A - ID_{B_0}$, where B_0 is the last mispredicted branch instruction. ② If ΔID is less than ID_THRESH (i.e., *A* and B_0 are very close in execution order), it suggests that the Icache miss of *A* is due to FTQ flushing caused by the branch misprediction of B_0 . As a result, we classify this Icache miss as branch misprediction-induced and identify B_0 as the src-branch for *A*. ③ Then, BTIP associates *A* and B_0 and updates the prefetch metadata based on the branch type of B_0 .

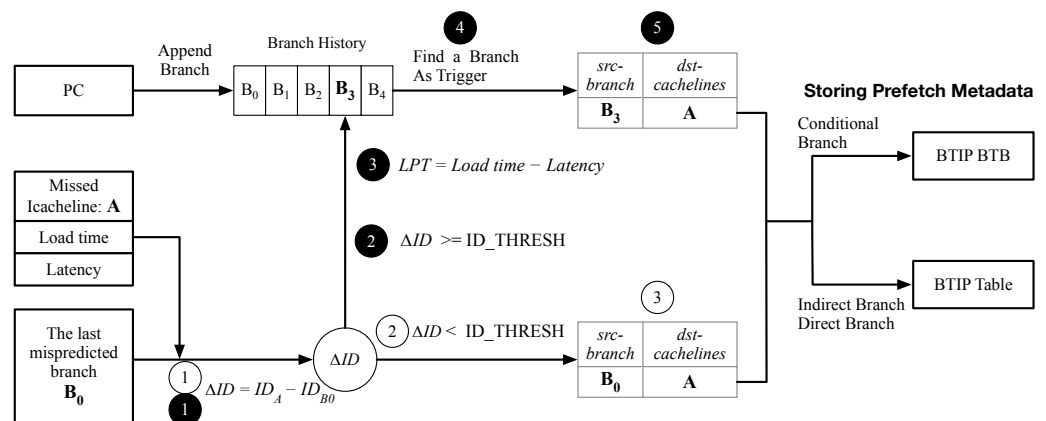


Figure 9. Steps for building and storing prefetch metadata.

Late Prefetch. ① As in the case above, BTIP calculates ΔID for each icache miss. ② If ΔID is greater than ID_THRESH , it indicates that FDIP initiated the prefetch for *A* early enough, but *A* still experienced an Icache miss. Therefore, we classify this miss as an Icache miss caused by late prefetch.

Next, BTIP identifies a suitable src-branch for the dst-cacheline. BTIP must determine how far in advance to issue a prefetch to avoid an Icache miss and which src-branch should be selected to meet this prefetch lead time. To achieve this, we extend the MSHR entry to track the fetch latency of A . When an Icache miss occurs, the MSHR allocates a new entry and logs the start timestamp, while the end timestamp corresponds to the cache fill time. The latency of the current Icache fetch is then calculated by subtracting the start timestamp recorded in the MSHR from the cache fill timestamp.

After measuring the fetch latency and load time (i.e., the start timestamp in the MSHR) of A , ③ we can calculate the latest prefetching timestamp (LPT) for A as *load time – fetch latency*. Next, BTIP needs to identify a src-branch that was executed before the LPT to prevent an Icache miss. To achieve this, we use a circular queue called the Branch History Queue (BHQ), which stores the most recently accessed branches and their corresponding access times. ④ BTIP scans the BHQ from the tail forward, selecting a branch whose access time is before the LPT to serve as the src-branch. If no suitable branch is found, the oldest branch in the BHQ is chosen as the src-branch. ⑤ Once the src-branch is identified, BTIP updates the prefetch metadata.

Storing the prefetch metadata. BTIP places the prefetch metadata in either the BTIP BTB or the BTIP Table based on the type of src-branch. In modern processors, BTB records branch targets for conditional and direct branches. Therefore, to reuse BTB information, we store the prefetch metadata for src-branches that are conditional branches in the BTIP BTB. Notably, BTIP does not store metadata for direct branches in the BTIP BTB. This is because BTIP-triggered prefetches only improve performance when a branch misprediction occurs. The only potential cause of misprediction for direct branches is a BTB miss. However, if a BTB miss occurs, BTIP cannot obtain the dst-cachelines needed for prefetching. Therefore, storing prefetch metadata for direct branches in the BTB is ineffective. Instead, we choose to store the prefetch metadata for direct branches in the BTIP Table, enabling BTIP to trigger prefetches even if a BTB miss occurs for a direct branch. For indirect branches, BTIP also records the prefetch metadata in the BTIP Table.

4.3. Triggering the Prefetches

BTIP attempts to prefetch each time the BPU generates a prediction result based on PC. During the prediction process, both the BTB and the BTIP Table are accessed simultaneously. If the BTB hits an entry in the BTIP BTB or the lookup in the BTIP Table results in a hit, it indicates that the PC corresponds to a src-branch. At this point, BTIP issues prefetch requests for the dst-cachelines stored in the hit entry.

FDIP and BTIP issue prefetch requests simultaneously, which can lead to performance interference. FDIP is a highly accurate prefetcher, and BTIP is designed to complement it. To avoid interfering with FDIP's prefetching requests, we assign higher priority to FDIP. Prefetch requests from the BTIP are processed only when the Icache has available read bandwidth, such as during IFU stalls or when the FDIP-PQ is empty.

4.4. The Modified BTB

BTIP saves storage overhead by reusing some fields of BTB entry such as PC_Tag and Target. It expands BTB entries by adding fields to store prefetch metadata. However, the number of BTB entries for modern processors is very large; e.g., AMD Zen5 [33] has 24 K BTB entries, and expanding the fields of each BTB entry would result in significantly higher storage overhead. Additionally, not all branches are src-branches. Therefore, we only expand some BTB entries to be BTIP BTB, allowing us to reuse existing BTB information without introducing excessive and unnecessary storage overhead.

Figure 10 shows the structure of our modified BTB and the composition of entries. We split the original BTB into two parts: the conventional BTB and the BTIP BTB. The conventional BTB remains unchanged from the original, while the BTIP BTB includes entries with extended fields (42 bits) to store prefetch metadata.

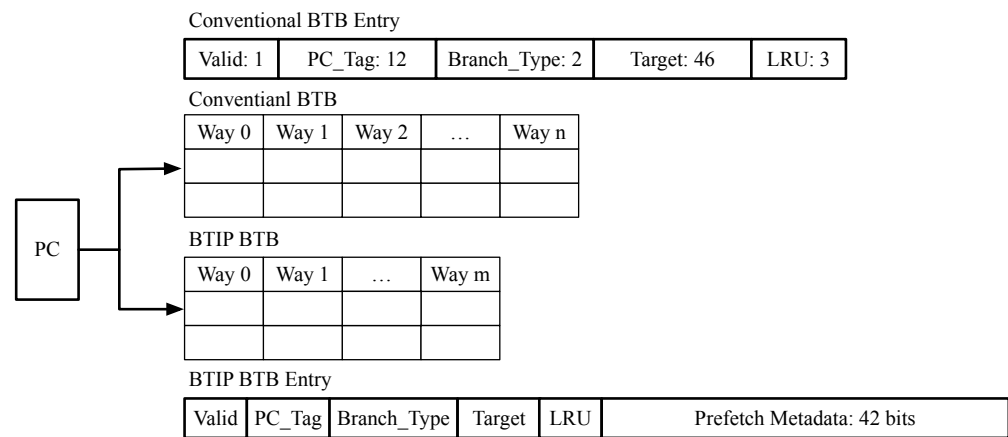


Figure 10. The modified BTB structure and entry composition.

BTB Lookup. When a PC enters the BPU, the BPU performs lookups on both the conventional BTB and the BTIP BTB. Our BTB update policy ensures that a PC will not have matching entries in both the conventional BTB and the BTIP BTB simultaneously. If the PC hits the conventional BTB, then the BPU takes the regular unfetched path. However, if the PC hits in the BTIP BTB, the BTB outputs additional prefetch metadata to the Prediction Unit, which then parses the metadata and extracts the destination address for prefetching.

The Update and Insert of Prefetch Metadata. When the relationship between src-branch and dst-cachelines is established, it needs to be stored in the BTIP BTB. At this point, the BPU uses the src-branch to perform lookups in both the conventional BTB and the BTIP BTB. If both lookups result in a miss, the BPU creates a new entry in the BTIP BTB and stores the prefetch metadata. If the BPU finds an entry in the conventional BTB, it selects an entry to evict in the BTIP BTB, swaps the fields between the two entries, and updates the prefetch metadata in the corresponding BTIP BTB entry. If the BPU finds an entry in the BTIP BTB, it directly updates the prefetch metadata of that entry.

4.5. Optimization

Filtering Useless Prefetches. BTIP may negatively impact performance when branch predictions are correct. To address this, we introduce a filtering mechanism to eliminate unnecessary prefetches. BTIP prefetch the Icache lines that were missed due to branch mispredictions. When a branch is mispredicted, BTIP's prefetching yields performance benefits. However, if the branch prediction is correct, BTIP may introduce unnecessary prefetches, potentially polluting the Icache. To alleviate the negative performance impact of excessive prefetching, BTIP filters out prefetches triggered by branches with high prediction accuracy.

This filtering mechanism relies on the estimated accuracy of each branch prediction. Specifically, BTIP tracks the hits and misses count for each prediction table of TAGE and ITTAGE. When TAGE and ITTAGE output branch prediction results, they also provide the estimated accuracy by dividing the misses count by the hits count. When the estimated accuracy is greater than the set threshold, BTIP discards the prefetch request.

Compressing the Prefetch Targets. A single src-branch can be correlated to multiple dst-cachelines. To maximize the number of dst-cachelines stored in metadata with limited length, we employ an address compression method using two key strategies. First, we reduce the storage consumed for each dst-cacheline by storing the address offset from the branch target rather than the full dst-cacheline address. This strategy works because the execution of a dst-cacheline typically occurs close in time to that of the src-branch, resulting in a small address offset between them. Consequently, saving address offsets instead of full dst-cacheline addresses effectively reduces the storage needed for each dst-cacheline.

Second, we adopt an efficient multi-address encoding approach to store as many dst-cachelines as possible. As shown in Table 1, we encode the dst-cachelines in four modes. We use 2 bits to represent the mode values and 42 bits to encode the address distances

of all corresponding dst-cachelines. Each mode allows for storing a different number of dst-cachelines that can fit within the 42-bit limit. For instance, in mode 0, the full 42 bits are used to store a single dst-cacheline, allowing for a maximum offset of $\pm 2^{41}$. In mode 2, the 42 bits are used to store four dst-cachelines, with each dst-cacheline occupying 10 bits, allowing for a maximum address offset of $\pm 2^9$. Since the metadata can have only one mode value, when a new dst-cacheline is inserted, we must recalculate the maximum offset to determine whether a mode change is necessary.

Table 1. Compression modes of dst-cachelines.

| Mode | Numbers of Dst-Cachelines | Width of Each Dst-Cacheline | Distance Range |
|------|---------------------------|-----------------------------|----------------|
| 0 | 1 | 42 | $\pm 2^{41}$ |
| 1 | 2 | 21 | $\pm 2^{20}$ |
| 2 | 4 | 10 | $\pm 2^9$ |
| 3 | 7 | 6 | $\pm 2^5$ |

4.6. Put it Together

L1I miss events and branch prediction operations trigger the BTIP to perform update metadata and issue prefetch operations, respectively. When an L1I miss occurs, the Prediction Unit retrieves miss information from the MSHR, including the missed Icacheline, load time, and latency. The Prediction Unit then determines whether the miss was caused by a branch misprediction or a late prefetch. If it was due to a branch misprediction, the previously mispredicted branch is selected as the src-branch. Otherwise, the Prediction Unit selects a branch executed early enough from the Branch History Queue as the src-branch. Finally, the src-branch and dst-cacheline are compressed and stored in the BTIP Table or BTIP BTB, completing the updating of metadata.

On the other hand, when the BPU generates a prediction based on the PC, BTIP predicts Icachelines that will be accessed in the future. BTIP uses this PC to look up both the BTIP BTB and BTIP Table. If the PC matches an entry in the BTIP BTB or BTIP Table, the Prediction Unit queries the TAGE Accuracy Table to obtain the estimated accuracy of the branch prediction. If the accuracy is below the predefined threshold, the Prediction Unit sends the Icachelines stored in the matched entry to the PQ to initiate a prefetch operation.

4.7. Storage Overhead

Table 2 details the storage overhead of BTIP, which consumes a total of 18.24 KB, which are allocated across its components as follows:

- The Branch History Queue (BHQ) is a 128-entry circular queue, where each entry contains a 64-bit field for the branch address and a 20-bit field for the execution timestamp. A 7-bit register points to the tail of the queue. The total memory required for the BHQ is 1345 bytes.
- The TAGE Accuracy Table is a 20-entry table where each entry records the hit and miss counts for TAGE and ITTAGE banks. Each entry consists of a 32-bit hit count and a 32-bit miss count. Every 10^9 cycles, the hit and miss counts are halved to prevent overflow. The TAT requires a total of 160 bytes.
- We modified the MSHR to record timing information for each Icache request, using a 16-entry structure with 12 bits to store the timestamp when the request was issued. This modification requires 24 bytes of memory.
- The BTIP Table is a 256-set, 4-way table, where each entry includes a 42-bit field for dst-cachelines, a 2-bit compression mode, and a 2-bit LRU indicator. The total memory required for the BTIP Table is 5888 bytes.
- We allocated 2048 entries from the conventional BTB to serve as the BTIP BTB. Each entry requires 42 bits for encoding dst-cachelines and 2 bits for compression mode. The total memory consumed by the BTIP BTB is 11,264 bytes.

Table 2. Details of BTIP’s storage overhead.

| Component | Storage |
|----------------------|---|
| Branch History Queue | $128 \times (64 + 20) + 8 = 10,760$ bits = 1345 bytes |
| TAGE Accuracy Table | $20 \times (32 + 32) = 1280$ bits = 160 bytes |
| MSHR | $16 \times 12 = 192$ bits = 24 bytes |
| BTIP Table | $1024 \times (42 + 2 + 2) = 63,488$ bits = 5888 bytes |
| BTIP BTB | $2048 \times (42 + 2) = 122,880$ bits = 11,264 bytes |
| Total | 18,681 bytes |

5. Evaluation and Discussion

5.1. Evaluation Setup

Simulation Parameters. We use a modified ChampSim [34] simulator to simulate and evaluate BTIP. We extended the default version of ChampSim to implement a recent state-of-the-art industry FDIP [15,35], BTB, TAGE [30] and ITTAGE [31]. The detailed simulator parameters are shown in Table 3.

Traces. We evaluate BTIP using the top 10 applications from IPC-1 that have the greatest optimization potential. We execute the first 100 million instructions as a warm-up, which is followed by the simulation of the next 100 million instructions.

Evaluated Prefetchers. We compared BTIP with four prior prefetchers:

- **FNL-MMA [36]:** FNL-MMA combines the Multiple Miss Ahead (MMA) prefetcher and the Footprint Next Line (FNL) prefetcher. The MMA prefetcher predicts future cache misses several steps in advance, allowing it to prefetch instruction blocks before they are needed. The FNL prefetcher predicts when the next cache lines will be accessed and selectively prefetches only those lines that are likely to be used soon.
- **D-JOLT [37]:** D-JOLT leverages both long-range and short-range prefetchers, where the long-range prefetcher predicts distant future memory accesses with broader coverage, and the short-range prefetcher provides more accurate predictions for near-future accesses. Additionally, a fallback prefetcher acts as a safety net when both long-range and short-range prefetchers fail.
- **PDIP [28]:** PDIP is designed to complement FDIP in modern processors. PDIP focuses on targeting cache misses that FDIP fails to hide, particularly those that cause front-end stalls after events like branch mispredictions.

Table 3. Simulation parameters.

| Front-end | |
|------------------------------|------------------|
| Fetch Width | 6 instructions |
| Decode Width | 6 instructions |
| Fetch Target Queue | 192 instructions |
| Decode Buffer | 32 instructions |
| Dispatch Buffer | 32 instructions |
| Branch Target Buffer | 16 K entries |
| Return Address Stack | 32 entries |
| Conditional Branch Predictor | 64 KB TAGE |
| Indirect Branch Predictor | 64 KB ITTAGE |
| Back-end | |
| Dispatch Width | 6 instructions |
| Execute Width | 4 instructions |
| Retire Width | 4 instructions |
| Re-order Buffer | 512 entries |
| Load Queue | 128 entries |
| Store Queue | 72 entries |

Table 3. Cont.

| Memory Hierarchy | |
|----------------------|------------------------------------|
| L1 Instruction Cache | 32 KB, 8-way, 4 hit cycles |
| L1 Data Cache | 32 KB, 8-way, 4 hit cycles |
| L2 Cache | 512 KB, 16-way, 10 hit cycles |
| L3 Cache | 1 MB, 16-way, 20 hit cycles |
| DRAM | 4 GB, one 64-bit channel, 800 MT/s |

5.2. IPC Performance

Figure 11 shows the IPC improvement of all evaluated prefetchers over a no-prefetcher baseline. The results show that all prefetchers had the performance improvement, and BTIP achieved the best performance improvement in almost all traces. Specifically, BTIP achieves an average IPC performance improvement of 49.4%, which is 5.1% higher than FDIP and 1.6% higher than PDIP. FNL-MMA and D-JOLT introduced complex prefetch mechanisms, achieving performance improvements of 33.8% and 32.1%, respectively. FDIP, leveraging highly accurate BPU predictions to prefetch instructions, outperformed both FNL and MMA with negligible storage overhead, delivering a 44.3% performance improvement. By prefetching additional Icache misses caused by branch mispredictions and late prefetches on top of FDIP, BTIP outperformed FDIP. Although the performance gains of PDIP and BTIP are similar, BTIP reduces storage overhead by 58.1% compared to PDIP by reusing BTB information.

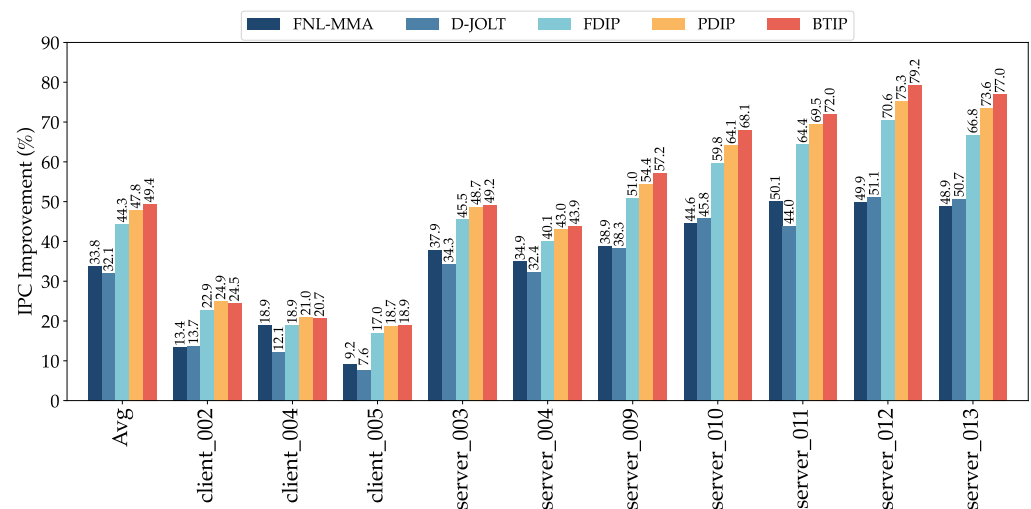


Figure 11. The IPC improvement of all prefetchers over no prefetch.

5.3. Icache Miss Reduction

Figure 12 shows the Icache MPKI of all evaluated prefetchers across all traces. The results demonstrate that BTIP significantly outperforms its competitors by substantially reducing the Icache miss. On average, BTIP reduced the Icache MPKI from 6.90 to 4.28 compared to FDIP. By optimizing Icache misses caused by both branch misprediction and late prefetching, BTIP reduces more Icache misses than PDIP when compared to FDIP. Although FNL-MMA and D-JOLT achieve lower Icache MPKI than FDIP, FDIP still delivers better overall performance. This is due to FDIP's decoupled front-end design, which allows the BPU to stay consistently ahead of the IFU. As a result, FDIP ensures that the application performance is not limited by the BPU's prediction bandwidth.

5.4. Prefetch Accuracy

Figure 13 shows the prefetch accuracy of all evaluated prefetchers across all traces. FDIP performs prefetching based on the BPU prediction results. Benefiting from the high prediction accuracy of the BPU, FDIP achieves the highest prefetch accuracy. PDIP and

BTIP build on FDIP by prefetching Icache lines missed due to branch mispredictions. However, since the prefetches issued by PDIP and BTIP are only effective when a branch misprediction occurs, both exhibit slightly lower prefetch accuracy compared to FDIP. In addition, BTIP extends PDIP by also prefetching Icache lines missed due to late prefetches. However, due to branch fan-out (single branch leading to many instructions) [7], the recorded src-branch does not always lead to the execution of the dst-cacheline, resulting in more useless prefetches compared to PDIP, thus reducing prefetch accuracy.

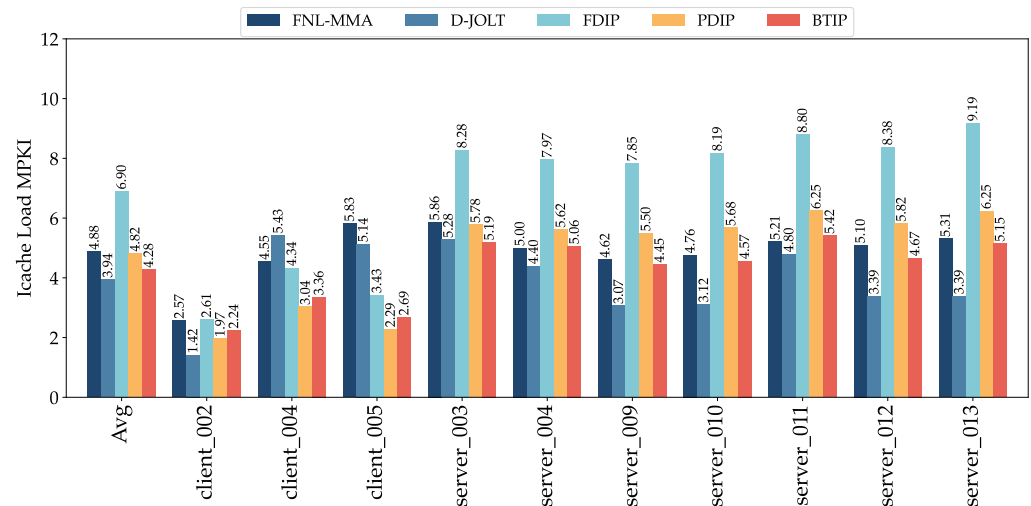


Figure 12. The Icache miss per kilo instructions.

5.5. Breakdown of Performance Improvements

In Section 3.2, our experiments revealed that late prefetching accounts for 52.48% of Icache misses, emphasizing the equal importance of optimizing both branch mispredictions and late prefetching. Therefore, BTIP is designed to address Icache misses caused by both factors. To more comprehensively evaluate the effectiveness of BTIP, we conducted a performance breakdown analysis.

Figure 14 illustrates the performance improvements achieved by BTIP. *BM* represents BTIP optimizing only the Icache misses caused by branch mispredictions, while *BM+LP* refers to optimizations targeting Icache misses caused by both branch mispredictions and late prefetching with FDIP as the baseline. The results in Figure 14 show that *BM* achieved a modest speedup of 1.6%, while *BM+LP* delivered a more substantial speedup of 3.5%. These findings demonstrate that BTIP effectively optimizes both sources of Icache misses.

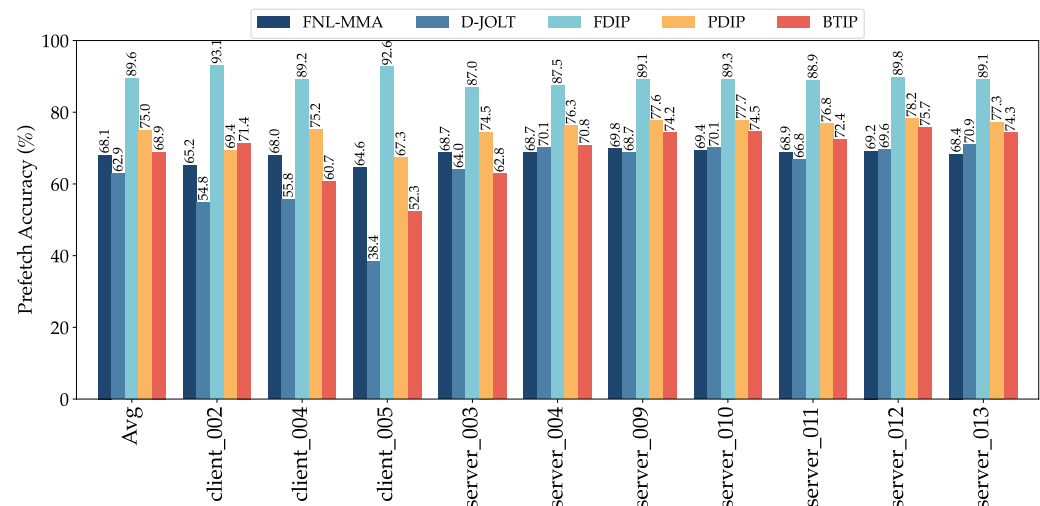


Figure 13. Prefetch accuracy.

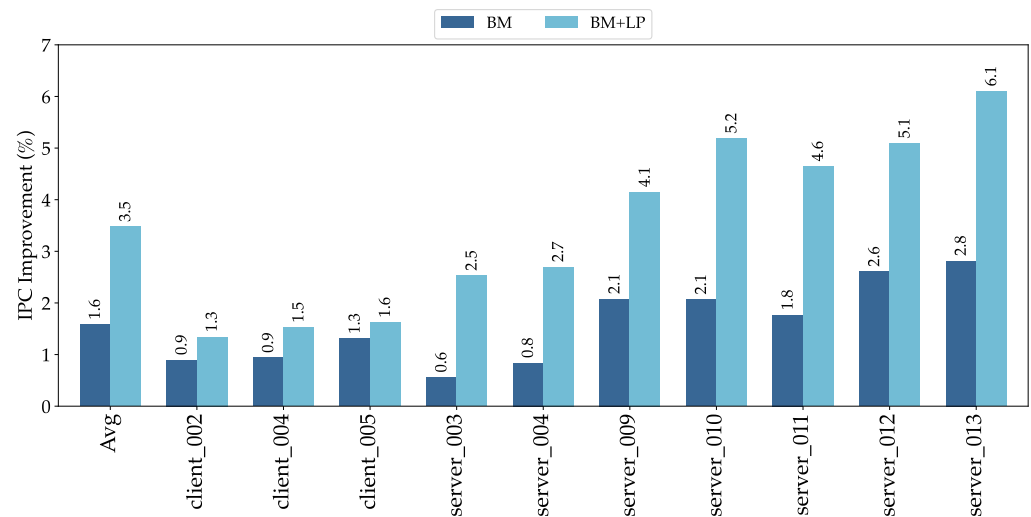


Figure 14. Breakdown of improvements on IPC. FDIP is the baseline.

5.6. Varying Icache Size

To evaluate the robustness of BTIP’s optimizations, we varied the Icache configuration to observe the impact of BTIP. Specifically, we measured the performance improvements of BTIP compared to FDIP with Icache sizes of 32 KB, 64 KB, 128 KB, and 256 KB, as shown in Figure 15. The results demonstrate that BTIP consistently achieves IPC gains compared to FDIP across all Icache sizes. The average performance improvements of BTIP over FDIP for 32 KB, 64 KB, 128 KB, and 256 KB Icache are 3.5%, 3.3%, 3.4%, and 2.8%, respectively.

From 32 to 128 KB, BTIP’s optimization effect remains stable with no significant decline. However, when the Icache size reaches 256 KB, BTIP’s performance gain slightly diminishes. This is because a 256 KB Icache is large enough to cover the code footprint of our test applications, reducing front-end bottlenecks and BTIP’s optimization potential. Nonetheless, this does not diminish BTIP’s value. On one hand, such a large Icache size introduces greater area and power consumption overhead as well as increased access latency. Furthermore, to our knowledge, no commercial processors currently feature such large Icache sizes. On the other hand, as mentioned in Section 1, application code footprints are growing year by year [8], and their growth rate exceeds that of Icache size increases. Therefore, we believe that even as Icache sizes continue to grow in future CPU iterations, BTIP will still provide noticeable performance improvements.

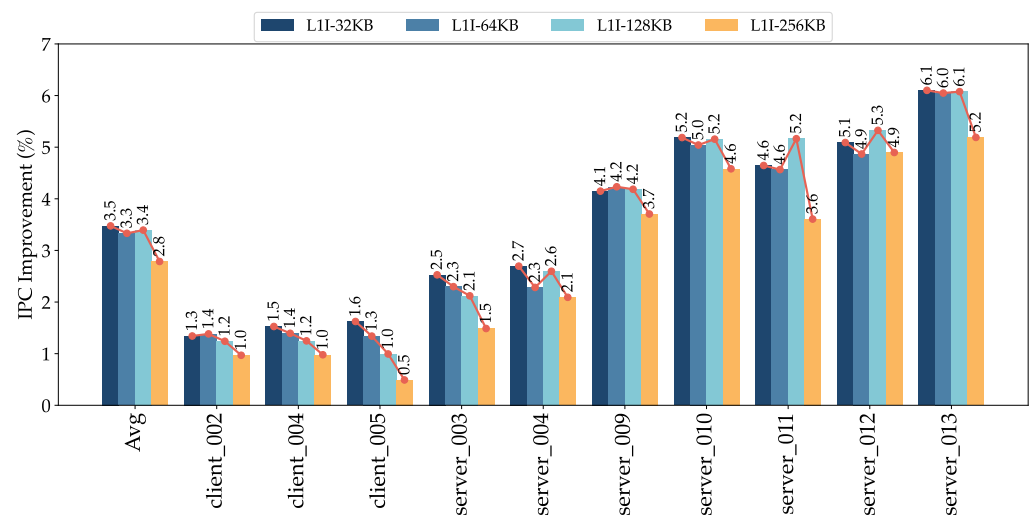


Figure 15. IPC Improvement of BTIP over FDIP with different Icache sizes.

5.7. Power and Area Analysis

We use Cacti 7.0 [38] to analyze the power consumption and area overhead of BTIP and PDIP. For BTIP, we calculate the power and area consumption of the BHQ, TAT, BTIP Table and BTIP BTB. For PDIP, we focused on the PDIP Table. The analysis was performed using 32 nm technology from Cacti 7.0, and the results were then scaled to 7 nm technology [39]. Power consumption for each application was calculated separately, and the average was taken as the final result.

The results are shown in Table 4. BTIP consumes 10.02 mW of power and 0.049 mm² area per core, which are, respectively, 78.22% and 38.58% of PDIP's values (12.81 mW and 0.127 mm²). Additionally, we compared BTIP's power and area overhead to those of existing 7nm CPUs. The results show that BTIP introduces minimal energy and area overhead. For instance, compared to the AMD Ryzen 5 5600X, BTIP only increases power consumption by 0.37% and area consumption by 0.09%. Therefore, we conclude that BTIP enhances performance with low power and area overhead.

Table 4. Power and area consumption of BTIP.

| BTIP Compared to Existing 7 nm CPUs | Power | Area |
|--|-------|-------|
| AMD Ryzen 5 5600X [40], 6 Cores, 65W TDP | 0.37% | 0.09% |
| AMD Ryzen 7 5800X [41], 8 Cores, 105W TDP | 0.49% | 0.08% |
| AMD Ryzen 9 5950X [42], 16 Cores, 105W TDP | 0.49% | 0.15% |
| AMD Ryzen Threadripper PRO 5975WX [43], 32 Cores, 280W TDP | 0.49% | 0.12% |
| AMD EPYC 7763 [44], 64 Cores, 280W TDP | 0.49% | 0.23% |
| Ampere Altra [45], 80 Cores, 250W TDP | 0.68% | 0.32% |
| BTIP's power: 10.02 mW/core. BTIP's area: 0.049 mm ² /core. | | |
| PDIP's power: 12.81 mW/core. PDIP's area: 0.127 mm ² /core. | | |

6. Discussion

In this section, we discuss the cost of BTIP and compare BTIP with FDIP and PDIP in terms of design, performance, power consumption, and storage (area) overhead.

6.1. Costs of BTIP

To implement BTIP, we extended and introduced several hardware components, including the TAGE Accuracy Table, BTIP Table, Branch History Queue, Modified MSHR, and BTIP BTB. While these components increase hardware storage (chip area) and power consumption, the resulting overhead is minimal. In particular, we detail the storage overhead of BTIP, totaling 18.24 KB in Section 4.7. In Section 5.7, we use cacti to calculate that these storage overheads take up a total of 0.049 mm² of the area and consume 10.02 mW of power on a 7 nm chip. The area and power consumption introduced by BTIP are negligible for existing CPUs. For instance, when implemented in the AMD Ryzen 5 5600X, BTIP results in only a 0.37% increase in power consumption and a 0.09% increase in area. Therefore, we can conclude that BTIP enhances performance with low power and area overhead.

6.2. Comparison with FDIP and PDIP

Modern high-performance processors employ decoupled front-end designs (e.g., FDIP), which actively prefetch Icache lines based on branch predictor control-flow predictions. While FDIP delivers significant performance gains with minimal hardware complexity, there is still considerable optimization potential, particularly in front-end intensive applications. Specifically, for the 10 most front-end intensive applications, FDIP has an average performance improvement potential of 14.7%. Our analysis identifies two main contributors to Icache misses in FDIP, branch misprediction and late prefetch, with 47.52% of misses caused by branch mispredictions and 52.48% by late prefetch.

PDIP [28] introduces a dedicated table to track Icache misses caused by branch mispredictions and trigger prefetch requests when the same branch is encountered. Although PDIP effectively reduces the number of Icache misses in FDIP due to branch mispredictions, it neglects Icache misses caused by late prefetch. Moreover, PDIP suffers from unnecessary high storage overhead due to storing too much duplicated information already stored in the BTB.

BTIP is designed to prefetch Icache lines that may miss due to both branch mispredictions and late prefetch. In addition, BTIP expands some BTB entries by adding fields to store prefetch metadata, enabling the reuse of existing BTB information without introducing excessive and unnecessary storage overhead. Consequently, BTIP achieves greater performance gains while requiring less storage than PDIP. Our evaluations show that BTIP outperforms FDIP and PDIP by 5.1% and 1.6%, respectively. BTIP requires only 18.24 KB of hardware storage compared to PDIP's 43.5 KB. Additionally, BTIP consumes 10.02 mW of power and 0.049 mm² of area per core, which are 78.22% and 38.58% of PDIP's power (12.81 mW) and area (0.127 mm²), respectively.

7. Related Works

In this section, we reviewed two types of instruction prefetchers: hardware instruction prefetchers and software instruction prefetchers.

7.1. Hardware Instruction Prefetcher

The next-line prefetcher [12], which prefetches consecutive Icache lines, is the simplest hardware spatial prefetcher. It is well optimized for continuous instruction streams but struggles with applications that have a significant amount of control flow.

Temporal instruction prefetching has gained attention for its ability to predict future cache accesses based on past misses. The Temporal Ancestry Prefetcher (TAP) [46] expands on this concept by approximating the transitive closure of a program's control flow graph, allowing it to predict long chains of instruction misses. TAP significantly outperforms next-line prefetchers in server workloads by effectively leveraging deep look-ahead in control flow graphs. PIF [47] improves instruction prefetching by utilizing the predictability of instruction streams while mitigating instability from microarchitectural factors like branch mispredictions and hardware interruptions. It records the correct-path, retire-order instruction stream, bypassing the noise introduced by wrong-path instructions and cache filtering. MANA [48] builds on this idea of leveraging temporal and spatial locality while focusing on reducing storage costs. It records compact metadata, grouping instruction blocks into spatial regions and linking them with successor pointers for efficient prefetching. Using a compact and efficient data structure, MANA achieves performance comparable to PIF but with 15.7× less storage overhead.

JIP [49] employs a hybrid approach by classifying instructions into non-branch, direct branch, and indirect branch categories, applying targeted prefetching strategies for each. JIP uses components like the Single Target Jumper (SJT) and Multiple Targets Jumper (MJT) to accurately prefetch future instructions. It is optimized for low hardware overhead by utilizing compressed address storage and temporal tables to enhance prefetch timeliness. RAS-directed instruction prefetching (RDIP) [50] records the return address of a function in the RAS along with the cache line of the Icache miss. When an RAS return occurs, RDIP checks whether the address change was recorded, and if so, prefetches the corresponding Icache line.

Several prefetchers were proposed in the 1st Instruction Prefetching Championship (IPC-1) [13,36,37]. EIP [13] introduces the concepts of code blocks and entanglement, recognizing consecutive blocks of code and linking them to the target code block where the control flow jumps. When an instruction in a source code block is encountered, EIP prefetches the entire source code block and the corresponding target code. It dynamically adjusts the prefetch look-ahead distance to account for cache miss latencies and application variations, ensuring timely prefetching. FNL+MMA [36] combines Footprint Next Line

(FNL) and Multiple Miss Ahead (MMA) strategies. FNL predicts whether the next cache line will be used in the near future, avoiding unnecessary prefetches, while MMA looks ahead to predict future misses. D-JOLT [37] addresses Icache misses by combining long-range and short-range prefetching techniques. It improves on RDIP with a new signature generation mechanism that uses an FIFO to store return addresses, rather than the stack used by RDIP, to generate prefetch targets. By integrating multiple prefetching strategies, D-JOLT adapts to varying distances between Icache misses, offering higher accuracy for short-term predictions and greater coverage for distant misses.

The Fetch-Directed Instruction Prefetching (FDIP) [14,15] decouples the branch predictor from the instruction cache to improve prefetching efficiency. This architecture allows the branch predictor to run ahead of the instruction cache, generating prefetch requests earlier and reducing instruction cache misses. PDIP [28] enhances instruction prefetching in FDIP. PDIP focuses on prefetching only those instruction cache misses that are critical to front-end performance and not effectively hidden by FDIP.

7.2. Software Instruction Prefetcher

Software prefetching techniques typically require modifications to the instruction set architecture so that Icache lines are prefetched in advance of their use. These techniques include inserting prefetching instructions into the code. Cooperative prefetching uses the compiler to statically analyze and automatically identify prefetched injection points. AsmDB [7] uses runtime profile information to find the critical Icache misses and insert prefetching instructions for these missed Icache lines to improve performance. I-SPY [32] also uses profile information, but it encodes the context information using a special directive to issue prefetches only when the context matches, thus reducing excessive prefetching when it is not needed. Also, I-SPY uses aggregate prefetching to reduce code bloat. Software prefetching techniques can be used to solve cold miss problems that cannot be solved by hardware techniques.

8. Conclusions

In this paper, we performed a detailed performance analysis of FDIP and found that it still suffers from Icache misses caused by branch mispredictions and late prefetching, leaving significant optimization potential. To address these issues, we proposed BTIP, a branch-triggered instruction prefetcher designed to handle Icache misses that FDIP cannot efficiently manage, specifically those caused by branch misprediction and late prefetch. To further minimize storage overhead, we introduced a novel BTB organization that stores prefetch metadata by reusing existing BTB entries. Our evaluation shows that BTIP outperforms FDIP and PDIP by 5.1% and 1.6% with only 41.9% of the storage overhead of PDIP.

Author Contributions: Funding acquisition, Y.C. and W.C.; Investigation, W.L.; Methodology, W.L., Y.L., S.C. and Y.Z.; Project administration, Y.C. and W.C.; Software, W.L.; Supervision, W.C.; Validation, Y.L., Z.J. and J.X.; Writing—original draft, W.L.; Writing—review and editing, Y.L., Y.C., Z.J. and J.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Alibaba Group through the Alibaba Innovation Research Program (No. 2024050953011).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare that this study received funding from the Alibaba Group. The funder had the following involvement with the study: design, analysis, and the decision to submit it for publication.

Abbreviations

The following abbreviations are used in this manuscript:

| | |
|--------|--|
| Icache | Instruction cache |
| FDIP | Fetch-Directed Instruction Prefetching |

| | |
|------|---|
| BTIP | branch triggered instruction prefetcher |
| PDIP | Priority-Directed Instruction Prefetching |
| BPU | Branch Prediction Unit |
| IFU | Instruction Fetch Unit |
| FTQ | Fetch Target Queue |
| BTB | Branch Target Buffer |
| PC | Program counter |
| RAS | Return Address Stack |
| MPKI | miss per kilo instructions |
| IPC | Instructions per Cycle |
| DRAM | dynamic random access memory |
| PQ | prefetch queue |
| MSHR | miss status holding register |
| TAT | TAGE Accuracy Table |
| BHQ | Branch History Queue |
| IT | Indirect Table |

References

1. Openjdk. Openjdk Main-Line Code Repository. Available online: <https://github.com/openjdk/jdk> (accessed on 8 September 2024).
2. VMware Tanzu. Spring Home. Available online: <https://spring.io> (accessed on 8 September 2024).
3. Google. gRPC. Available online: <https://grpc.io> (accessed on 8 September 2024).
4. Apache Software Foundation. Apache Log4j 2. Available online: <https://logging.apache.org/log4j/2.x/> (accessed on 8 September 2024).
5. Lin, W.; Qin, J.; Chen, Y.; Jin, Z.; Xu, J.; Zhang, Y.; Cai, S.; Fu, L.; Chen, Y.; Chen, W. JACO: JAVa Code Layout Optimizer Enabling Continuous Optimization without Pausing Application Services. In Proceedings of the 2023 IEEE International Conference on Cluster Computing (CLUSTER), Santa Fe, NM, USA, 31 October–3 November 2023; pp. 295–306. [\[CrossRef\]](#)
6. Ferdman, M.; Adileh, A.; Kocberber, O.; Volos, S.; Alisafae, M.; Jevdjic, D.; Kaynak, C.; Popescu, A.D.; Ailamaki, A.; Falsafi, B. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *Acm Sigplan Not.* **2012**, *47*, 37–48. [\[CrossRef\]](#)
7. Ayers, G.; Nagendra, N.P.; August, D.I.; Cho, H.K.; Kanev, S.; Kozyrakis, C.; Krishnamurthy, T.; Litz, H.; Moseley, T.; Ranganathan, P. AsmDB: Understanding and mitigating front-end stalls in warehouse-scale computers. In Proceedings of the 46th International Symposium on Computer Architecture, Phoenix, AZ, USA, 22–26 June 2019; pp. 462–473. [\[CrossRef\]](#)
8. Kanev, S.; Darago, J.P.; Hazelwood, K.; Ranganathan, P.; Moseley, T.; Wei, G.Y.; Brooks, D. Profiling a warehouse-scale computer. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015; pp. 158–169. [\[CrossRef\]](#)
9. Panchenko, M.; Auler, R.; Nell, B.; Ottoni, G. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, 16–20 February 2019; pp. 2–14. [\[CrossRef\]](#)
10. Panchenko, M.; Auler, R.; Sakka, L.; Ottoni, G. Lightning BOLT: Powerful, fast, and scalable binary optimization. In Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction, Virtual, Republic of Korea, 2–3 March 2021; pp. 119–130. [\[CrossRef\]](#)
11. Ayers, G.; Ahn, J.H.; Kozyrakis, C.; Ranganathan, P. Memory Hierarchy for Web Search. In Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria, 24–28 February 2018; pp. 643–656. [\[CrossRef\]](#)
12. Baer, J.L. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors*; Cambridge University Press: Cambridge, UK, 2009.
13. Ros, A.; Jimborean, A. A Cost-Effective Entangling Prefetcher for Instructions. In Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 14–18 June 2021; pp. 99–111. [\[CrossRef\]](#)
14. Reinman, G.; Calder, B.; Austin, T. Fetch directed instruction prefetching. In Proceedings of the MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture, Haifa, Israel, 16–18 November 1999; pp. 16–27. [\[CrossRef\]](#)
15. Ishii, Y.; Lee, J.; Nathella, K.; Sunwoo, D. Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective. In Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Stony Brook, NY, USA, 28–30 March 2021; pp. 172–182. [\[CrossRef\]](#)
16. Kaynak, C.; Grot, B.; Falsafi, B. Confluence: Unified instruction supply for scale-out servers. In Proceedings of the 48th International Symposium on Microarchitecture, Waikiki, HI, USA, 5–9 December 2015; pp. 166–177. [\[CrossRef\]](#)
17. Kumar, R.; Grot, B.; Nagarajan, V. Blasting through the Front-End Bottleneck with Shotgun. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, USA, 24–28 March 2018; pp. 30–42. [\[CrossRef\]](#)

18. Chen, I.C.K.; Lee, C.C.; Mudge, T.N. Instruction prefetching using branch prediction information. In Proceedings of the Proceedings International Conference on Computer Design VLSI in Computers and Processors, Austin, TX, USA, 12–15 October 1997; IEEE: New York City, NY, USA, 1997; pp. 593–601.
19. Kumar, R.; Huang, C.C.; Grot, B.; Nagarajan, V. Boomerang: A Metadata-Free Architecture for Control Flow Delivery. In Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), Austin, TX, USA, 4–8 February 2017; pp. 493–504. [[CrossRef](#)]
20. Pierce, J.; Mudge, T. Wrong-path instruction prefetching. In Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29, Paris, France, 2–4 December 1996; pp. 165–175. [[CrossRef](#)]
21. Spracklen, L.; Chou, Y.; Abraham, S. Effective instruction prefetching in chip multiprocessors for modern commercial applications. In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA, 12–16 February 2005; pp. 225–236. [[CrossRef](#)]
22. Srinivasan, V.; Davidson, E.S.; Tyson, G.S.; Charney, M.J.; Puzak, T.R. Branch history guided instruction prefetching. In Proceedings of the Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture, Monterrey, Nuevo Leon, Mexico, 19–24 January 2001; IEEE: New York City, NY, USA, 2001; pp. 291–300.
23. Adiga, N.; Bonanno, J.; Collura, A.; Heizmann, M.; Prasky, B.R.; Saporito, A. The ibm z15 high frequency mainframe branch predictor industrial product. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020; IEEE: New York City, NY, USA, 2020; pp. 27–39.
24. Grayson, B.; Rupley, J.; Zuraski, G.Z.; Quinnell, E.; Jiménez, D.A.; Nakra, T.; Kitchin, P.; Hensley, R.; Brekelbaum, E.; Sinha, V.; et al. Evolution of the samsung exynos cpu microarchitecture. In Proceedings of the 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), Valencia, Spain, 30 May–3 June 2020; IEEE: New York City, NY, USA, 2020; pp. 40–51.
25. Pellegrini, A.; Stephens, N.; Bruce, M.; Ishii, Y.; Pusdesris, J.; Raja, A.; Abernathy, C.; Koppanalil, J.; Ringe, T.; Tummala, A.; et al. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro* **2020**, *40*, 53–62. [[CrossRef](#)]
26. Pellegrini, A. Arm Neoverse N2: Arm’s 2 nd generation high performance infrastructure CPUs and system IPs. In Proceedings of the 2021 IEEE Hot Chips 33 Symposium (HCS), Palo Alto, CA, USA, 22–24 August 2021; IEEE: New York City, NY, USA, 2021; pp. 1–27.
27. Rupley, J.; Burgess, B.; Grayson, B.; Zuraski, G.D. Samsung m3 processor. *IEEE Micro* **2019**, *39*, 37–44. [[CrossRef](#)]
28. Godala, B.R.; Ramesh, S.P.; Pokam, G.A.; Stark, J.; Seznec, A.; Tullsen, D.; August, D.I. PDIP: Priority Directed Instruction Prefetching. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, La Jolla CA USA, 27 April–1 May 2024; Volume 2, pp. 846–861.
29. NC State University. The 1st Instruction Prefetching Championship. Available online: <https://research.ece.ncsu.edu/ipc/welcome/> (accessed on 9 September 2024).
30. Andre, S.; Pierre, M. A case for (partially) TAGged GEometric history length branch prediction. *J. Instr.-Level Parallelism* **2006**, *8*, 23.
31. Seznec, A. A 64-Kbytes ITTAGE indirect branch predictor. In Proceedings of the JWAC-2: Championship Branch Prediction, JILP, San Jose, CA, USA, 4 June 2011.
32. Khan, T.A.; Sriraman, A.; Devietti, J.; Pokam, G.; Litz, H.; Kasikci, B. I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing. In Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020; pp. 146–159. [[CrossRef](#)]
33. Wikipedia. AMD Zen 5. Available online: https://en.wikipedia.org/wiki/Zen_5 (accessed on 9 September 2024).
34. Gober, N.; Chacon, G.; Wang, L.; Gratz, P.V.; Jimenez, D.A.; Teran, E.; Pugsley, S.; Kim, J. The Championship Simulator: Architectural Simulation for Education and Competition. *arXiv* **2022**, arXiv:2210.14324. [[CrossRef](#)]
35. Ishii, Y.; Lee, J.; Nathella, K.; Sunwoo, D. Rebasing Instruction Prefetching: An Industry Perspective. *IEEE Comput. Archit. Lett.* **2020**, *19*, 147–150. [[CrossRef](#)]
36. Seznec, A. The Fnl+ Mma Instruction Cache Prefetcher. The 1st Instruction Prefetching Championship. Available online: <https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/FNLMMA-final.pdf> (accessed on 9 September 2024).
37. Nakamura, T.; Koizumi, T.; Degawa, Y.; Irie, H.; Sakai, S.; Shioya, R. D-Jolt: Distant Jolt Prefetcher. The 1st Instruction Prefetching Championship. Available online: <https://research.ece.ncsu.edu/ipc/> (accessed on 9 September 2024).
38. Balasubramonian, R.; Kahng, A.B.; Muralimanohar, N.; Shafiee, A.; Srinivas, V. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* **2017**, *14*, 1–25. [[CrossRef](#)]
39. Stillmaker, A.; Baas, B. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration* **2017**, *58*, 74–81. [[CrossRef](#)]
40. x86 CPU’s Guide. AMD Ryzen 5 5600X Specs. Available online: <https://www.x86-guide.net/en/cpu/AMD-Ryzen-5-5600X-cpu-no8125.html> (accessed on 22 October 2024).
41. x86 CPU’s Guide. AMD Ryzen 7 5800X Specs. Available online: <https://www.x86-guide.net/en/cpu/AMD-Ryzen-7-5800X-cpu-no8127.html> (accessed on 22 October 2024).
42. x86 CPU’s Guide. AMD Ryzen 9 5950X Specs. Available online: <https://www.x86-guide.net/en/cpu/AMD-Ryzen-9-5950X-cpu-no8130.html> (accessed on 22 October 2024).

43. x86 CPU's Guide. AMD Ryzen Threadripper PRO 5975WX Specs. Available online: <https://www.x86-guide.net/en/cpu/AMD-Ryzen-Threadripper-PRO-5975WX-cpu-no8356.html> (accessed on 22 October 2024).
44. x86 CPU's Guide. AMD EPYC 7763 Specs. Available online: <https://www.x86-guide.net/en/cpu/AMD-EPYC-7763-cpu-no8262.html> (accessed on 22 October 2024).
45. Ampere Computing. Ampere Altra Family Product Brief. Available online: <https://amperecomputing.com/briefs/ampere-altra-family-product-brief> (accessed on 22 October 2024).
46. Gober, N.; Chacon, G.; Jiménez, D.; Gratz, P. Temporal Ancestry Prefetcher. The 1st Instruction Prefetching Championship. Available online: https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/tap_final.pdf (accessed on 9 September 2024).
47. Ferdman, M.; Kaynak, C.; Falsafi, B. Proactive instruction fetch. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, Porto Alegre, Brazil, 3–7 December 2011; pp. 152–162. [CrossRef]
48. Ansari, A.; Golshan, F.; Barati, R.; Lotfi-Kamran, P.; Sarbazi-Azad, H. MANA: Microarchitecting a Temporal Instruction Prefetcher. *IEEE Trans. Comput.* **2022**, *72*, 732–743. [CrossRef]
49. Gupta, V.; Kalani, N.S.; Panda, B. Runjump-Run: Bouquet of Instruction Pointer Jumpers for High Performance Instruction Prefetching. The 1st Instruction Prefetching Championship. Available online: <https://research.ece.ncsu.edu/wp-content/uploads/sites/19/2020/05/JIP.pdf> (accessed on 9 September 2024).
50. Kolli, A.; Saidi, A.; Wenisch, T.F. RDIP: Return-address-stack directed instruction prefetching. In Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, Davis, CA, USA, 7–11 December 2013; pp. 260–271. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.