INCORRECT-PATH PREDICTION FOR POWER THROTTLING SPECULATIVE EXECUTION

by

Naomi A. Rehman

Baskin School of Engineering
University of California, Santa Cruz
March 2025

Table of Contents

List of Figures

List of Tables

Abstract

Incorrect-Path Prediction for Power Throttling Speculative Execution

by

Naomi A. Rehman

Today's processors have achieved remarkable performance. In the past decades we have seen great performance increases, despite the emergence of physical limitations to transistor scaling. To achieve this CPUs are now very speculative, reducing the performance limitations of dependencies by doing work before all information is available. While this technique is essential to performance, it also has performance and power implications. Specifically, when speculation goes wrong, it can hurt performance and dramatically increase the total power required to run a program. In this work, we examine a mechanism to predict when speculation has gone wrong, and use this information to reduce its negative impact. We focus primarily on power saving, and show that significant amounts of power can be saved. Our implementation outperforms the prior work, Pipeline Gating, in some scenarios, however, further refinement is needed to see power savings without significant performance degradation.

Acknowledgments

First, I'd like to thank my advisor, Prof. Heiner Litz, for his continuous support and guidance on this project, and for introducing me to computer architecture. Second, I'd like to thank my second advisor and thesis committee member, Prof. Tyler Sorensen, for his excellent and thorough feedback. Third, I'm grateful to my final thesis committee member, Prof. Dustin Richmond, for his guidance and expertise on this project. Finally, I'm grateful to my friends and family who have supported me throughout my research.

Chapter 1

Introduction

1.1 Motivation

A key factor in the performance and efficiency of a CPU is speculative execution. Modern processors have very deep and wide pipelines [56] [15], allowing several instructions can be in flight simultaneously. However, the pipeline is typically not fully utilized due to hazards or dependencies in the application that would be violated if the instructions issued without stalling. Control hazards, which exist in the presence of control flow instructions, prevent the pipeline from being filled because the Program Counter (PC) of the next instruction is not known until the control flow instruction has resolved (i.e. finished executing). As the distance between control flow instructions is typically lower than the maximum number of instructions that can be in the pipeline [15], control flow instructions pose a serious problem for processor performance.

Modern CPUs leverage Branch Predictors (BP) [5,5,13,17,18,23,28,29,33,34,36,36,

38,44–48,51,53,57,58] and Branch Target Buffers (BTB) [1,1,6,8,11,24,25,27,28,37,49,54] to keep pipelines full in the presence of unresolved branches . These structures predict the outcome of a control flow instruction by predicting the direction (taken or not taken) and target (next PC if taken), respectively. If both predict correctly, the processor will continue fetching correct, or on-path, instructions, effectively hiding the stalls that would have occurred were there no BP or BTB. When the BP or BTB predict incorrectly, however, the processor executes down the *wrong-path*, executing instructions that will never retire. Once the mispredicted control flow instruction resolves, all younger instructions must be flushed from the pipeline, and the work spent on them is wasted.

Wrong-path execution results in substantial performance and energy overheads. Fig. 1.1 shows the percent decrease in cycles with a perfect branch predictor, perfect BTB and a combination of the two for the SPEC2017 benchmark suite. A significant portion of cycles, in some cases over 50%, are wasted due to branch mispredictions and BTB misses. This is because it takes several cycles, in some cases over 1,000 (Figure 1.2), to detect and recover from a misprediction. The high misprediction penalty means even small increases in BP accuracy can have large impacts IPC. On average about 10% of cycles can be saved with a perfect BP and BTB, corresponding to a power reduction of approximately the same magnitude.

Besides wasting compute cycles and power, wrong-path instructions can have additional adverse side-effects such pollution of micro-architectural structures such as caches, incurring additional performance and energy efficiency degradation [41]. The performance impact of such issues depends on the application, as some workloads can benefit from off-path prefetches [35].

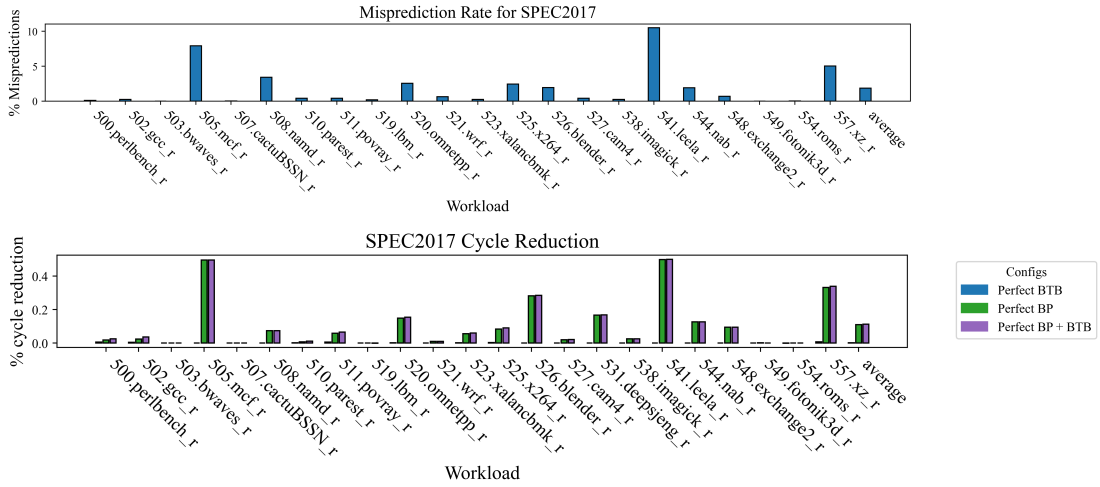

Figure 1.1: Percent cycle reduction of a simulated processor with perfect BP, BTB and a combination of both. Results were obtained simulating a Sunny-Cove like architecture [56] in the Scarab simulator [35]

To address these issues, previous work has focused mainly on improving the accuracy of the BP [5, 17, 18, 23, 33, 34, 36, 38, 42, 44–47, 51, 53, 57], improving the organization of the BTB [1, 1, 6, 8, 11, 24, 25, 27, 28, 37, 49, 54], or reducing the penalty of wrong-path execution [2, 3, 10, 14, 16, 31]. These works have shown considerable performance gains, however, they also have considerable storage requirements. For example, the most recent winner of the branch prediction championship (CBP), TAGE-SC-L, consumes 64KB of storage and BTBs have also grown to that size. Not only is this incredibly expensive in terms of area and power, but increasing the size of these structures further is challenging as they must have low latency lookups, typically in the range of 1-2 cycles. Additionally, further improving branch predictor accuracy is difficult as many of the remaining mispredictions are hard to predict [12, 59], meaning they aren't strongly correlated with execution history, and BPs struggle to learn their behavior. These prior works mainly focus on improving performance, and few methods have

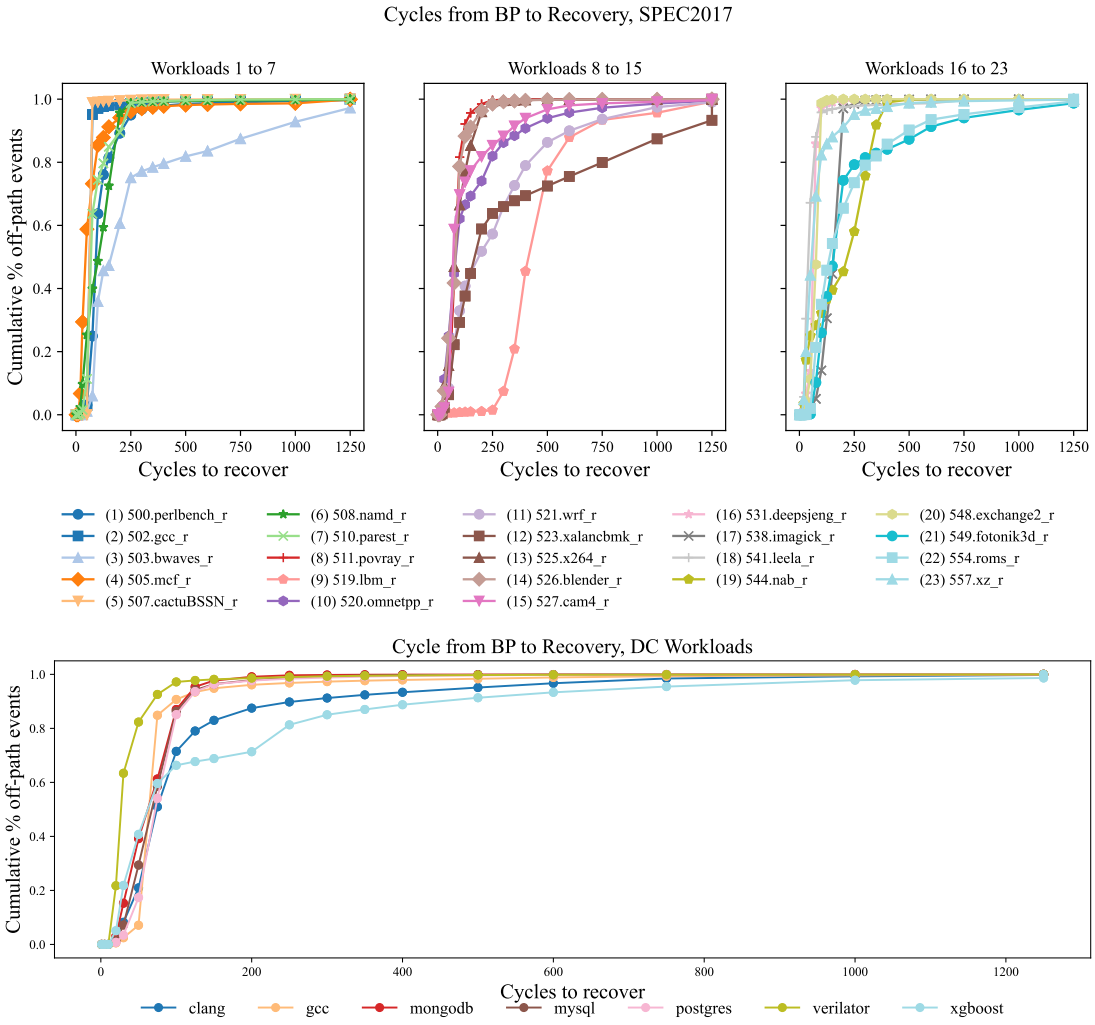

Figure 1.2: Cycles to recover from and off-path event. Most events resolve in between 5 and 200 cycles, however some take over 1000 cycles to resolve.

aimed to directly address the power implications of erroneous speculation.

To address these issues, we propose incorrect-path prediction. Incorrect-path prediction does not aim to predict the outcome of individual branches; instead, it predicts for each cycle whether the CPU is likely on or off-path. This approach is based on the following insights:

1. It is still useful to detect off-path execution after a misprediction or BTB miss has occurred, i.e. a low precision off-path predictor is still useful.

2. More information becomes available in the cycles after a misprediction occurs that can be used to predict off-path execution. For example, a sequence of low confidence branches may indicate a misprediction occurred recently.

Accurate incorrect-path prediction can be exploited for several optimization opportunities, including powering down the frontend to reduce energy and throttling inaccurate prefetch requests.

1.2 Contributions and Outline

Our contributions are as follows:

1. A proposed new approach to improving CPU frontends: incorrect path prediction.

2. An evaluation of our implementation against a previous mechanism for reducing power consumption from bad speculation.

Chapter 2 describes the background and related literature for this work. Chapter 3 contains our approach for this work, and Chapter 4 contains an evaluation of our approach

compared to existing work. Chapter 5 details potential future directions for this work.

Chapter 2

Background

2.1 CPU Frontends

The *frontend* is the component of a CPU tasked with providing instructions to the *backend*, or execution units. Today's processors are pipelined, meaning the execution of an instruction is split into several stages. Processors today have anywhere from 2 to 20 pipeline stages [15, 56], depending on the intended application. Intel's high performance architectures have between 14-19 pipeline stages, allowing several instructions to be in flight at once [56]. While pipelining is a huge contributor to the high performance of modern processors, it introduces some challenges. A key challenge posed by pipelining is determining which instruction to execute next. In the absence of control flow this decision is trivial; simply execute the next instruction in program order. Control flow makes this more difficult, as the next PC is unknown until the control flow instruction has been executed. In a single cycle processor this is trivial, but with pipelining it takes several cycles for the outcome of a branch instruction to be determined

after it is fetched and decoded. In those cycles the next address of instructions is unknown. To avoid stalling, processors predict the PC of the next instruction.

There are two components of predicting the next PC; predicting the direction of the branch (taken or not taken), and predicting the target of the branch. The first is accomplished with the branch predictor, and the second with the branch target buffer (BTB). Current branch predictors learn application behavior to accurately predict whether a given branch will be taken or not taken. The BTB stores the targets of branches, allowing the frontend to continue fetching ops after a taken branch without decoding to find the next PC.

The information available at prediction time depends on whether the prediction happens before or after decoding the instruction. More about the instruction is known after decode; for example, an unconditional indirect branch contains all information needed to continue fetching subsequent instructions is known. However, the penalty of waiting to predict branches until after they are decoded is too high for this to be done in practice. Instruction fetch and decode together comprise several pipeline stages; predicting after decode would necessitate either a) stalling for each instruction to decode, as a control flow instruction can only be detected after decode and any instruction could be control flow, or b) speculatively decode instructions on the fallthrough path and resteer fetch to the target PC after a branch is predicted taken. The first option starves the subsequent pipeline stages, as the frontend would only produce an instruction every N cycles, where N is the number of stages for fetching and decoding an instruction. The second option will lead to frequent frontend flushes, as most control flow instructions are taken, causing a similar issue of few instructions being provided to the backend.

This is clearly a serious performance issue, so branch prediction and BTB lookup

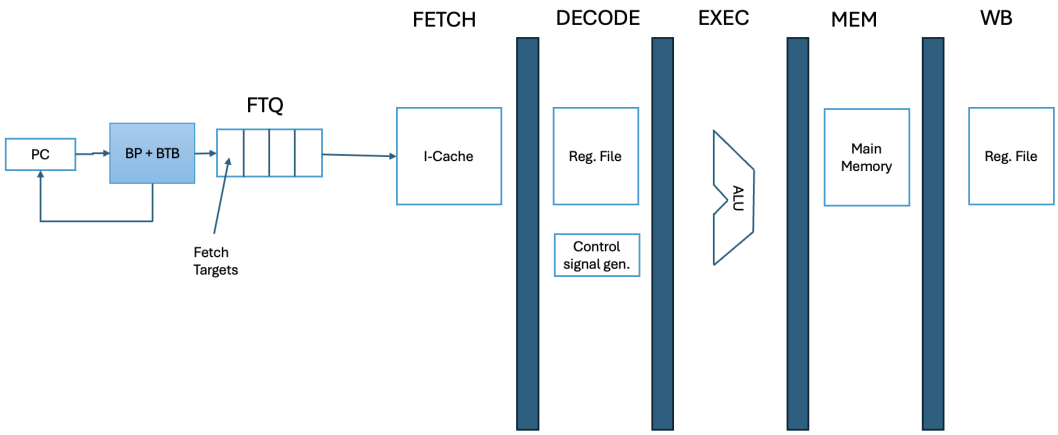

Figure 2.1: Decoupled frontend architecture. The BP runs ahead of the rest of the processor, producing fetch targets consumed by the Fetch stage.

happen before instruction fetch in most processors. This is called a *decoupled frontend* (Figure 2.1) [39, 40]. In such an architecture, the instruction stream is predicted and used to create a queue of fetch targets (FTs), referred to as the Fetch Target Queue (FTQ). As the instruction type is not known at this stage, all instructions are looked up in the BP and BTB. On a taken branch the current FT is ended and pushed to the FTQ, and a new FT is started, similar to a basic block. This allows the frontend to run ahead of the rest of the pipeline, producing blocks to be fetched and decoded without relying on the backend to resolve control flow.

The main drawback of moving branch prediction earlier in the pipeline is that it increases the misprediction penalty, flushing more instructions per misprediction. Speculative execution is not always correct, and when it goes wrong the processor must flush all speculative work to maintain program correctness. We define three reasons for incorrect speculative execution, or *wrong-path* execution:

1. Mispredictions: The branch predictor predicted the wrong branch outcome, e.i. taken

when the branch should not have been taken or not taken when the branch should have been taken.

2. BTB Misses: The target of the control flow instruction was not present in the BTB or IBTB.

3. Misfetches: The outcome of the control flow instruction was correctly predicted by the BP, but the target was incorrectly predicted (aliasing in BTB or IBTB, incorrect IBTB branch PC, return address stack underflow).

We refer to any one of these as an off-path event, and the instruction that experiences one of these events a trigger instruction. All trigger instructions will eventually induce a recovery when detected by the backend. All instructions younger than the trigger instruction will be flushed from the pipeline upon recovery. While mispredictions can only be resolved after a branch is executed, BTB misses and misfetches can be resolved at decode as the target becomes available at this stage.

The performance of a processor is tightly coupled with the accuracy of the frontend. If the frontend is not producing the correct instruction stream, all subsequent stages of the pipeline will spend several cycles performing useless work. The exact number of cycles to resolve a branch instruction is not fixed, and depends on several factors, including the depth of the pipeline, the latency of the prior instructions, and which stage the delay is being measured from. Figure 1.2 shows the distribution of the cycles to resolve a branch after BP. For most workloads, half of off-path events are resolved after 50 cycles, and the majority of remaining events are resolved between 50-200 cycles after branch prediction. This is why even small

improvements to frontend accuracy; in half the cases, an off-path event takes over 50 cycles to resolve.

2.2 Power Implications of Time Spent Off-Path

Besides the obvious performance loss, we identify two key issues with cycles spent executing off-path instructions. The first issue, and the issue we primarily focus on in this work, is power consumption. The power consumption of a processor is defined as [55]:

$$P_{static} = V * I \tag{2.1}$$

$$P_{static} = P_{shortcircuit} + \alpha C V^2 f \tag{2.2}$$

$$P_{total} = P_{static} + P_{dynamic} \tag{2.3}$$

Static power is inherent to the layout and technology of a chip, and is expended for the full time the chip is powered. In contrast, dynamic power depends on the frequency the chip is run at and the switching frequency, which quantifies how often gates switch from on to off and vice versa. Switching expends power because a) it takes power to change the voltage of a gate and b) when a transistor switches it briefly short circuits and sees a spike in current. Off-path execution doesn't impact static power, but contributes to higher dynamic power consumption. As off-path work is never used, all the power spent producing it is wasted.

2.3 Micro-architectural Pollution from Time Spent Off-Path

The second issue with off-path execution is the potential for polluting structures in the processor. Off-path prefetches to the I and D-cache can evict useful entries, leading to higher miss rates and longer access times [41]. However, this effect is application dependent, as some applications actually benefit from off-path prefetches. Due to this complication, in this work we do not focus on addressing this effect of off-path cycles, however, prior work has shown that our mechanism combined with other techniques can provide benefit for such workloads [35].

2.4 Challenges to Improving Frontend Accuracy

As detailed in prior sections, it is important to both performance and efficiency to reduce the time spent off-path. There are currently three main challenges in improving frontend accuracy: timing requirements, hard-to-predict branches, and increasing application size. Each of these is discussed in the following sections.

2.4.1 Timing Requirements

One strategy to improve frontend accuracy is making the BTB and BP larger, so they can store more program information. A larger BP can store more PC-History outcomes, and a larger BTB can store more targets. Unfortunately, this is not straightforward, as the lookup latency is limited to 1-2 cycles. Longer latency lookups impact performance by effectively reducing the frontend bandwidth. Lookups can be pipelined to hide long latencies, however, if one of these branches is predicted to be taken, the frontend essentially performs a 'mini'

flush, discarding the predictions for the instructions following the taken branch and looking up addresses starting at the target PC. After this mini flush, the frontend will need to wait the full latency of the BP lookup to continue producing fetch targets. Without fetch targets the fetch stage stalls, and subsequent stages are starved of instructions. This latency is also exposed after a backend resteer, where the FTQ is flushed and the frontend will not produce an FT for at least the full latency of the BP lookup, which will starve the fetch stage for that many cycles.

2.4.2 Hard-to-Predict Branches

Branch predictors are already very good at what they do, and cover a majority of branches in programs. On the SPEC benchmark suite the 2016 Branch Prediction Championship winner, TAGE-SC-L, achieved an average of 2.71 MPKIs on the BPC-5 traces [46]. The remaining branches not covered by current branch predictors are hard-to-predict, meaning they have properties that make them difficult to predict based on the combination of history and PC most BPs leverage. Many of these are data-dependent branches, where the outcome of the branch depends on data loaded from memory. These are typical of pointer chasing structures, such as trees. In a balanced tree, like a red-black tree, the probability of a branch being taken or not taken is equal, and depends on the key being searched and the value of the most recent node visited. The only pattern to learn is the probability of a branch being taken or not taken, which can be biased if certain keys are looked up more frequently. With a self-balancing tree, even this is difficult as the structure of the tree changes.

Consider the binary search tree lookup defined in Figure 2.4.2. The outcome of the branches on lines 2 and 4 depend on the node key value, which must be loaded from mem-

```
1 def tree_lookup(key, node):
2         if(node.key == key):
3                 return node.val
4        if(key < node.key):
5                tree_lookup(key, node.left)
6        else:
7                tree_loopup(key, node,right)
```

Figure 2.2: Tree lookup.

ory, and the value of the key being looked up. There is no deterministic pattern for what the outcomes of these branches will be. The load which feeds a data-dependent branch is not inherently problematic to prediction; the branch predictor would not (in most predictors) use the load value regardless of whether it was in a register or came from memory. Rather, the load implies the computation happened very far back in the branching history, and thus the limited history length used by the BP can't take correlated branches involved in producing that value into account.

2.4.3 Increasing Application Size

Recent work from Google profiling datacenter, or *warehouse scale*, computers shows that their search binaries are increasing significantly in size, as much as 27% year on year [4,20]. Their research finds that the size of the binary itself can lead to performance degradation. The larger working set of instructions can be too large to fit into the I-Cache, BP and BTB, leading

to frequent BTB misses and mispredictions.

In general, as the application binary size increases, frontend accuracy decreases. This can be in part addressed by larger BPs and BTBs. Indeed, modern chips tend towards larger branch predictors and BTBs, with the current storage budget for a high performance BP being around 64KB [46]. However, the extra latency introduced from the larger capacity eventually cancels out any benefit, preventing massive BPs and BTBs from being implemented to support these applications. Additionally, such large BPs and BTBs are very expensive, and this extra area cost is wasted on applications that are small enough to fit into current predictive structures.

2.5 Past Strategies for Reducing Off-path Cycles

Despite these challenges, there have been significant efforts to improve the accuracy of the frontend. The main strategies to attain this have been:

1. Improve branch predictor accuracy.

2. Improve BTB organization.

3. Reduce misprediction penalty.

To our knowledge there is only one other work which predicts off-path cycles and uses this for power throttling [32], which is over two decades old. While there is little work directly related to our proposed mechanism, we include the related works in the areas mentioned above as they provide context for the current challenges and efforts in frontend architecture.

2.5.1 Improving BP Accuracy

Improving BP accuracy is the most obvious strategy for reducing off-path cycles. In moderately sized applications, such as the SPEC2017 benchmark suite [7], mispredictions on conditional branches are a large contributor to off-path cycles (fig cycle reduction), and present great opportunity for improving performance. For example, SPEC17 benchmarks 505.mcf_r and 541.leela_r both see approximately 10% of branches mispredicted in a realistic simulation with a TAGE-SC-L branch predictor [46], but see 50% cycle reduction with a perfect branch predictor.[1]

State-of-the-art branch predictors typically use some form of online learning to dynamically learn application branching behaviors. The two most common learning algorithms are TAGE [47] and Perceptron [18]. Both are simple enough to implement in hardware, keeping storage and training overheads acceptable for a real processor. TAgged GEometric History Length (TAGE) predictors use an array of lookup tables for prediction. These tables are indexed with a hash of the global branch history and the PC of the branch being predicted. Global history is effectively a shift register updated on each branch, where a zero is shifted in on a not taken branch and a one is shifted on a taken branch. In a TAGE predictor, each table is indexed by a different geometric history length. These tables contain saturating counters that are incremented when a branch is taken and decremented when the branch is not taken. Each entry includes a tag to avoid aliasing between branches [47].

[1] To understand the dramatic effect of reducing mispredictions consider a processor where it takes one cycle to execute a correctly predicted instruction and 20 cycles to resolve a misprediction. If an application consisted of 100 branches, and had a 99% BP accuracy, it would take 99 + 20 = 119 cycles to execute the program. Removing the single misprediction, it would now take 100 cycles to execute the program, a 17% performance improvement for a 1% BP accuracy improvement.

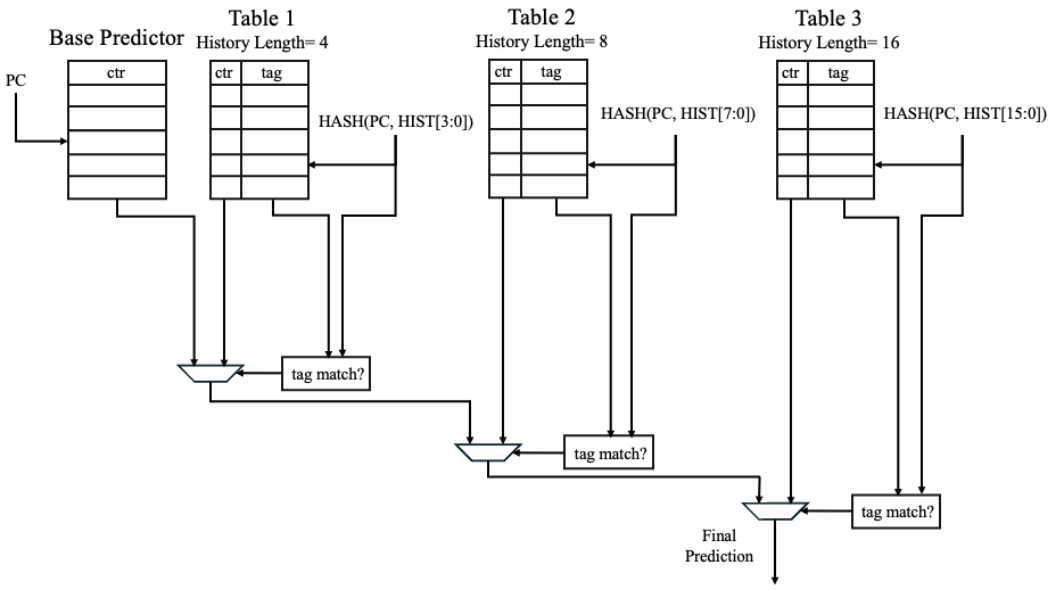

Figure 2.3: The TAGE Predictor. Several separate tables contain tagged prediction counters. Each table is indexed by a hash of PC and different geometrically increasing history lengths. A base predictor, provides the prediction if a branch misses in all other tables.

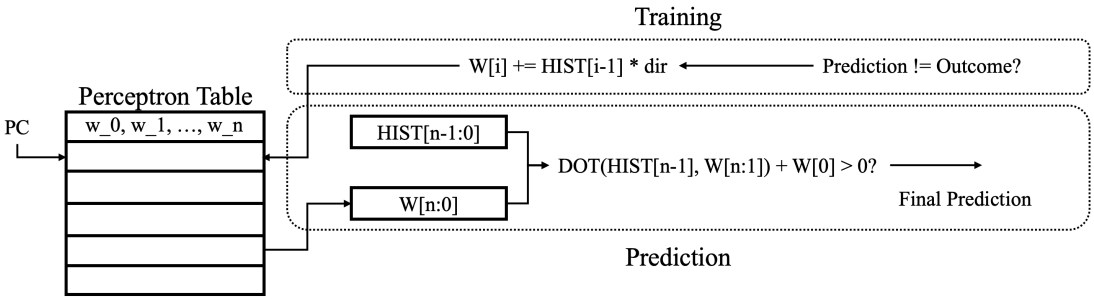

Figure 2.4: Basic perceptron branch predictor. A table of weights is looked up by the branch address and trained with the branch outcome.

A key challenge to TAGE-like predictors is that the number of entries necessary to uniquely identify PC-history pairs grows exponentially with the number of history bits. In contrast, perceptron predictors, based on the perceptron ML algorithm, grow their storage requirements linearly with global history length [18]. These predictors use the global history as features, and a table of weights is indexed by the PC of the branch being predicted. These weights can learn which bits of the history are highly correlated with the branch outcome, and can ignore the noise of uncorrelated branches in the history. Branches not correlated with the outcome will have a (very roughly) equal number of taken and not-taken outcomes relative to the predicted branch, so their weights will stay close to zero and not contribute to the prediction. The main limitation of perceptron is that it can't learn linearly-inseparable functions, such as XOR.

Current branch predictors employ a mix of predictors, an arrangement called a tournament predictor [15]. This allows weaknesses of one predictor to be covered by the strengths of another. The winner of the most recent CBP winner, TAGE-SC-L, includes a TAGE predictor, a loop predictor, and a statistical corrector. The loop predictor tracks fixed-length loops, and

overrides the TAGE prediction to cover the loop termination. The statistical corrector tracks whether this combination of history, PC, and confidence has been mispredicted by TAGE, and can be used to invert the TAGE prediction.

More recent implementations of Perceptron are hashed perceptrons which index tables of weights with a hash of the PC and global history, rather than just the PC [51]. In this adaptation, the features are not the outcome of individual branches in the history, but the outcome of a hash of the PC and global history. Each weight is summed to produce the final prediction. Hashed perceptrons are more robust to non-linearity than the original perceptron because the weights of a given prediction come from several different hashes. The most recent Hashed Perceptron predictor, the Multi-perspective Perceptron, uses several tables of weights indexed by different representations of the branch history, which are summed to produce the final prediction. Its main contribution over the original hashed perceptron is the addition of additional features, or ways of, representing the branch history.

TAGE and Perceptron-based predictors are very accurate. Modern branch predictors typically have <5 Mispredictions Per Kilo Instructions (MKPI), better than 99% accuracy. Despite this high accuracy, mispredictions remain a significant contributor to reduced IPC. The remaining instructions not covered by TAGE-SC-L and similar predictors are considered *hard-to-predict* branches [59]. Even with infinite storage budgets these branches are not well-covered by the SOTA predictors [46], and have characteristics that make them difficult to predict based purely on history and PC.

A significant portion of these remaining branches are *data-dependent* branches, meaning they are dependent on a recent load from memory. These branches are typical of graph-like

structures, such as linked lists and binary search trees. While a PC-History pair may be statistically biased due to application patterns, it's very challenging to accurately predict the outcome as there is no precise pattern to be learned. Several recent works in branch prediction aim to address this challenge.

One such work, Address-Branch Correlation, aims to address this by predicting data-dependent branches using the load address [12]. This work takes advantage of the fact that many data structures don't change for long periods of execution. For example, in a linked list, the terminating condition of the traversal is dependent on the value of the last node. If the structure of the linked list hasn't changed, the address of the node determines the branch outcome. In the general case, if the branch outcome is dependent on structure, not value, it can be predicted based on address. ABC learns which load addresses consistently correspond to a following branch outcome, improving IPC by 6.3% over a 16KB TAGE.

A more recent work, LDBP, also aims to improve data-dependent branches. LDBP is more aggressive than ABC [50]. Instead of learning the outcome based on addresses, it 'prefetches' the data needed for the branch computation, then uses that value to predict the branch outcome. It leverages the Stride data prefetcher, a popular data prefetcher that learns the strides between memory accesses. On branch completion, LDBP trains the Stride prefetcher to fetch the load address the branch depends on, and sets a bit to indicate this load is used for branch prediction. This method works when the load address is predictable, for instance in linked lists.

Another subset of hard-to-predict branches are those with noisy, non-deterministic histories. For example, a branch correlated with another several instructions earlier may have a

variable number of uncorrelated branches in between. Each different history would be allocated a separate entry in TAGE and Perceptron, which can increase aliasing with other branches and training times, as two entries need to be trained. It's worth noting that the original PC-indexed perceptron can't learn this pattern at all, as the contribution of a branch to the prediction is strictly tied to its position in the global history. The Multi-perspective Perceptron addresses this with Modulo-History, which uses slices of the history to remove some of the noisy, non-deterministic parts of the history [19]. As the slices taken is not a trainable parameter, this technique can also remove useful parts of the history, although these may be covered by other tables of weights in the predictor.

A more robust approach to this class of branches uses off-line ML techniques to identify and predict them. Tarsa et al. propose the use of ML helper predictors to assist the main branch predictor at runtime [52]. They implement CNNs to learn correlations between hard-to-predict branches and previous branches in the history. Off-line training allows the CNN helper to consider much longer history lengths than are practical to train in an on-chip predictor. The trained model is used at run-time using a small CNN on-chip inference engine. Branch-Net builds on this work with a more thorough analysis and different CNN structure that provides improved performance over the original work of Tarsa et al [59].

Inferencing off-line trained ML models at runtime is a subset of a technique known as profile-guided optimization (PGO), which leverages offline analysis to improve performance of programs. This is not limited to branch prediction, and can be used to optimize instruction ordering in the binary, insert software prefetching into the binary, inform cache replacement policies, etc. Whisper also leverages PGO to improve BP accuracy. Instead of training a neural

network, Whisper finds a boolean formula that relates the history to the outcome of a static branch. Whisper operates on hashed (compressed) histories by creating a fixed-length hashed history from histories of geometric lengths [22]. The learned boolean formula is encoded into the binary with a **brhint** instruction, which is used at run-time to determine the direction of the branch associated branch. Whisper requires fewer on-chip resources compared to Branch-Net and achieves a slightly better MKPI reduction over TAGE-SC-L than Branchnet.

2.5.2 Improving BTB Accuracy

IPC speedup is not always directly correlated to BP improvements, as branch direction is only half of the information needed to continue executing. If the branch predictor correctly predicts a branch as taken but the target is not present in the BTB, the prediction is useless since the target PC is unknown. In this case, the processor will incorrectly continue to execute down the fall-through path. Thus, BTB efficacy is a critical component of frontend performance.

In some sense, BTB organization is a much simpler problem than branch prediction. Except in the case of indirect branches, the target PC of a branch is deterministic, so complex pattern matching is not required. The BTB is effectively a cache, and benefits from similar techniques used to optimize caches: hierarchy, replacement policies, and prefetching.

Hierarchical BTBs are useful for balancing the need for a large working set of branch targets and low latency BTB lookups. In such BTB organizations, a small first-level BTB provides a target quickly. On a tag mismatch, the larger secondary BTB is looked up [24]. While this is effective, multi-level BTBs require huge amounts of expensive on-chip storage. To address this issue, Burshea and Murshovos propose a virtualized BTB, where parts of the BTB

are stored in the on-chip caches rather than in the BTB itself, allowing for a larger BTB without increasing the physical storage budget [9]. Their work, Phantom-BTB, maps the second-level BTB to a reserved portion of the physical address space, which can be allocated in the L2 cache. As the entire second level BTB does not need to be physically mapped at all times, the chip can dynamically allocate entries in the L2 cache to the BTB when necessary, and used for instructions and data in other cases. This work reduces the high hardware overhead of large BTBs, although it must contend with the high latency of retrieving second-level entries from the L2.

Another approach to reducing BTB misses is BTB prefetching. Similar to cache prefetching, rather than keeping all the targets on chip they are prefetched before they are needed, which allows high accuracy low-latency lookups and reduces on-chip storage overhead. Some BTB prefetching mechanisms are implemented entirely in hardware. The most recent of these, Shotgun, uses the majority of the BTB storage budget to track unconditional branches [26]. This is based on the insight that global execution of an application is spatially divided into smaller regions entered by unconditional control flow, such as function calls. The cache blocks within these regions have good spatial locality, typically lying within just a few cache lines of the unconditional branch target. By storing the target and associated spatial positions of cache-lines within the region, Shotgun can prefetch lines that will soon be used and predecode them to fill the BTB.

The main limitation of Shotgun is that it can only prefetch instructions within a small window of the unconditional control flow. Conditional branches outside of this region will not be prefetched and will still miss in the BTB. Twig, in contrast, leverages PGO to identify and

address branches that frequently miss in the BTB [21]. It first identifies the branch PCs that consistently miss in the BTB. It then modifies the application binary to include a new proposed instruction, `brprefetch`, which contains the branch PC and its target. This instruction prefills the target into the BTB, ensuring the subsequent branch does not miss. To reduce instruction footprint overhead, Twig also introduces a `brcoalesce` instruction, which prefetches multiple PC-target pairs stored in memory.

2.5.3 Reducing Misprediction Penalty

The third body of work aimed to reduce time spent off-path looks at reducing the misprediction penalty. Within this body, one strategy has been to identify instructions that will be executed regardless of the branch outcome, or control-flow independent instructions. Consider an if-else statement. Regardless of the outcome of the branch, the code following the if-else statement will eventually execute. Skipper aims to hide some of the misprediction penalty by identifying low-confidence branches and stopping fetch for everything before the predicted merge-point, and all instructions dependent on what occurs in the possibly mispredicted region [10]. This allows the processor to continue making progress regardless of the branch outcome. Skipper's main weakness is that if it incorrectly identifies a branch as a misprediction it will prevent useful instructions from executing, which can reduce IPC more than it helps. Its successor, Ginger, also aims to exploit control flow independence, but takes a different approach. Ginger dispatches all instructions after a potential misprediction, but flushes only the instructions that are control-flow dependent [16].

Other work attempts to recover the pipeline early after a misprediction. Gupta et al.'s

work aims to resteer early after data-dependent branches [14]. In their work, they introduce a class of branches they call direct data dependent (3D) which are branches with only one feeding load and very few instructions in the chain between the load and branch. They track these 3D branches and use pre-computation to determine the actual outcome of the branch shortly after its predicted by the branch predictor.

While there is only one other work which predicts off-path cycles and uses this prediction for power throttling, there are a few other works that are similar in concept to ours. The first, Wrong Path Events (WPE), uses unusual behaviors in execution to detect and recover from mispredictionsm [3]. This is based on the insight that after a misprediction, some instructions following the misprediction may be executed before the actual branch is executed. These instructions can consume uninitialized register values, leading to unusual events. For example, a pointer may be initialized to 0, and the address set in the mispredicted region. If an instruction consumes this pointer value it will trigger a null-pointer exception that would not have occurred were the processor on-path. They use such unusual events as an indicator that the processor has gone off-path and use this to recover early by predicting which branch mispredicted. When the branch can't be determined, they gate fetch to prevent more off-path instructions from executing.

Another paper, Branch Misprediction Prediction, proposes a complementary branch predictor. This predictor predicts mispredictions to reverse the BP prediction [43]. It tracks the distance between two mispredictions and the address of the latter misprediction. The main structure is the mispredicted branch table (MPBT), which is indexed by a concatenation of `PC XOR global history`, the misprediction history, and the number of branches between the

last two mispredictions. While their results were very good on SPEC2000 benchmarks, this work evaluated against now outdated branch predictors such as gshare with very modest storage budgets. They show that a gshare predictor would need >64KB storage budget to match their 4KB storage budget predictor, however, most predictors today are in the 64KB range, and it's unclear if this methodology would continue to provide benefit over today's branch predictors. Both this work and WPE both differ from ours in a key way: we do not attempt to predict the misprediction PC or recover, so we can show benefit with much lower accuracy.

The final work similar to our proposed mechanism is Manne and Grunwald's work, Pipeline Gating [32]. They propose tracking the number of low-confidence branches in the pipeline to dynamically throttle fetch and decode. The concept of this work is very similar to ours, however we propose a more detailed predictor that leverages several signals to produce a prediction.

Chapter 3

Incorrect-Path Prediction

We propose a mechanism, Incorrect-Path Prediction (I2P), to predict each cycle whether the processor is likely to be on the wrong-path. Due to the deep and wide pipelines of modern CPUs, it often takes several cycles for the backend to resolve a misprediction. Since covering some of these cycles is still useful for clock gating and prefetch throttling, I2P can have low accuracy and still be useful. Figure 1.2 shows the number of cycles required to resolve a misprediction. If the mechanism predicts the misprediction accurately, it can mitigate microarchitectural pollution and power consumption by throttling-prefetching and clock-gating the frontend.

3.1 Intuition

This mechanism can provide benefits even with relatively low prediction accuracy, which is demonstrated in Figure 3.1. We refer to the region of cycles between two resteers as a resteer window. Within the window, at some point there will be a misprediction, and subsequent

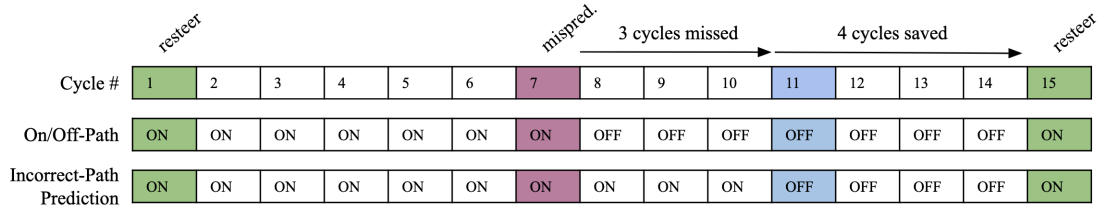

Figure 3.1: On and off-path cycles in a misprediction window. Cycles following the misprediction are off-path until the resteer. Despite being late, the I2P prediction can still clock gate for four cycles.

cycles will be off-path. In the example the misprediction happens in cycle 7, and seven cycles are off-path before the resteer, where the frontend is redirected to the correct next PC and subsequent instructions are on-path. If the incorrect-path predictor detects this misprediction before the backend, the next cycles can be used for clock gating. In the above example the I2P predictor predicts the misprediction in cycle 10, and cycles 11-14 are 'caught', and can be used for clock gating.

This allowable lag between the actual misprediction and the I2P prediction allows I2P to utilize additional features that may become available after misprediction. For example, if several low-confidence branches have entered the pipeline, there is a high probability that one of them was mispredicted. Our technique differs from existing work that utilizes complex second-level predictors to override initial predictions with early resteers in two ways:

1. Our approach does not have a fixed latency; it continuously aggregates information to refine its prediction.

2. It utilizes information (such as branch predictor history) that is generated after the branch misprediction.

3.2 Perceptron Implementation

We have implemented an online learning mechanism that leverages the perceptron algorithm. This learns weights for the following subset of the features mentioned above, which we have found to be the most beneficial for the perceptron implementation. We do not include several of the features found to be useful in the feature analysis due to their continuous values. Features with a range of possible values, rather than binary values, increase the hardware cost of implementing perceptron considerably as they require multiplication hardware. In our experiments we found these additional features added very little performance benefit over implementations using only the features in Table 3.1.

The predictor consists of a table of weights, indexed by the XOR of the global history and the branch PC modulo the number of entries in the table. Each entry stores the weights for that particular history-PC pair. The entries are untagged, as this reduces the storage requirements considerably. Weights are 8-bit saturating counters, and each weight vector includes a bias weight.

The I2P prediction is performed directly following branch prediction. At prediction time, the weights for that history-pair are looked up and multiplied by the current feature values. Most of the features are binary, represented as a -1 false and 1 for true, so the multiplication step is trivial and does not require large, dedicated multiplication hardware. The sign of the sum of these features is used as the prediction, with a positive sum indicating off-path and a negative sum indicating on-path.

The training algorithm for the weight table is detailed in Figure 3.2. In the traditional

Feature	**Description**
FT Length	Length in bytes of the current FT
I-cache boundary	Is the current FT ended by an I-cache boundary
Taken Branch	Is the current FT ended by a taken branch
TAGE Base	Was the prediction provided by the TAGE Base component
TAGE Short	Was the prediction provided by the TAGE Short component
TAGE Long	Was the prediction provided by the TAGE Long component
TAGE Loop	Was the prediction provided by the TAGE Loop component
TAGE Stat. Corrector	Was the prediction provided by the TAGE Stat. Corrector component
Branch Confidence	Confidence of the current prediction (hysteresis bits of TAGE ctr)

Table 3.1: Micro-architectural features used for I2P prediction.

```
1    def train_i2p(PC, history, prediction, outcome, features):
        weight_table_idx = (PC ^ history) % N_ENTRIES
2       weights = weight_table[weight_table_idx]
3        if (prediction == outcome)
4            return
5        dir = outcome == off-path ? 1 : -1
6        for i in range(len(weights)):
7            weights[i] += dir * sign(features[i])
```

Figure 3.2: I2P training algorithm. Weights are stored in a table indexed by a hash of PC and history, and updated if the prediction differs from the actual outcome.

perceptron algorithm, weights are incremented by `diff * learning_rate * feature`, where `diff` is the difference between the correct label and the prediction. I2P instead increments weights by `dir`, which considerably simplifies the hardware as no multication is needed.

Training occurs when the actual outcome of a prediction becomes available, i.e. when a branch retires or the FTQ is flushed. This mechanism cannot be trained purely on retired branches, as it would never see examples of off-path control flow, and would only learn to predict on-path. To account for this, we train on resolved branches and off-path branches from the FTQ. On a resteer, FTQ entries are usually flushed. To train our mechanism on these branches, FTs are copied out of the FTQ into a training buffer on a flush, and drained in the following cycles to train the weight table.

In our implementation, we choose to use the first off-path prediction as a switch, and continue predicting off-path until a) the backend encounters a misprediction and triggers a resteer or b) the instruction that triggered the off-path prediction resolves. Returning I2P to

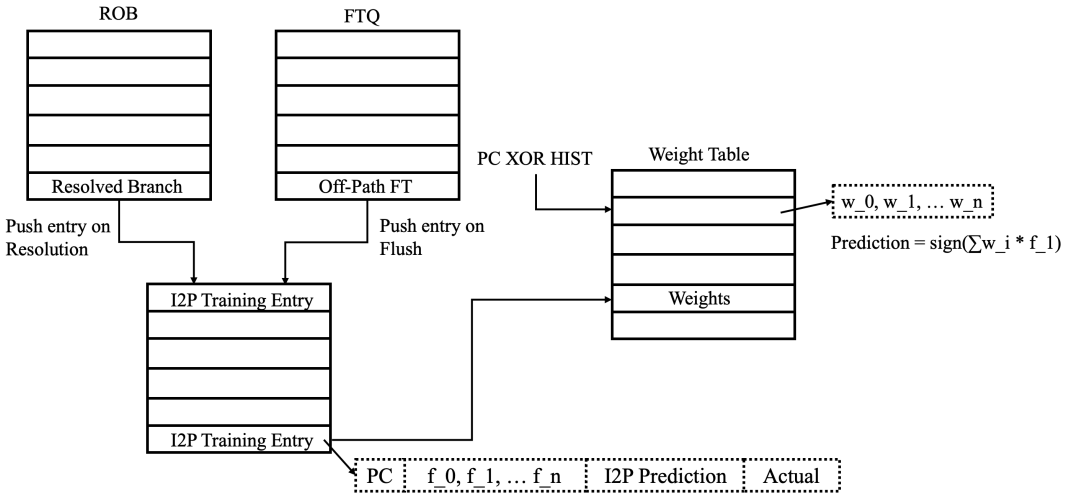

Figure 3.3: I2P Perceptron Implementation. I2P maintains a table of weights for each PC-History hash.

predicting on-path after the trigger instruction resolves reduces the penalty of predicting off-path too early and prevents deadlock when the prediction is used to clock-gate the frontend.

With each weight being represented as an 8-bit saturating counter, nine features, and 512 entries in the table, we estimate the storage requirements of our design to be 4 KB.

Chapter 4

Evaluation

4.1 Simulation Methodology

We simulate our mechanism with Scarab, an open-source cycle-accurate CPU simulator [35]. Scarab is capable of both trace-driven and execution-driven simulation. In execution driven mode, Scarab executes instructions from the original application binary through Intel's PIN tool [30]. In trace-driven mode, Scarab executes a stream of instructions previously collected with DynamoRIO from the an execution of the application binary. Scarab realistically simulates off-path execution in both modes. The only inaccuracy in trace-driven mode is that it does not simulate register values, thus off-path loads and stores use replayed addresses. If on the wrong-path scarab fetches a PC not found in the trace, it will treat the instruction as a NOP.

We configure Scarab in a deep, wide out-of-order pipeline based on Intel's Sunny-Cove architecture [56]. The simulated processor has a decoupled frontend architecture with an FDIP instruction prefetcher. The full list of parameters can be found in Table 4.2.

Workload	Description
Clang	Compiler
GCC	Compiler
MongoDB	Database Program
MySQL	Database Program
Postgres	Database Program
Verilator	Verilog Simulator
XGBoost	Gradient Boosted Forest (Inference)

Table 4.1: Datacenter workloads.

We evaluate I2P on the SPEC2017 benchmark suite, as well as several workloads which were chosen for their large instruction working sets. These larger workloads (Table 4.1) are intended to emulate the challenges posed by the large instruction footprints of modern datacenter applications [35].

4.2 Accuracy Metric

A key consideration of our performance metric was ensuring our mechanism is evaluated independently from the performance of the BP and BTB. A standard interpretation of accuracy, total correct cycles out of total cycles, would be biased towards a high accuracy for workloads where the branch predictor was already very accurate, as our mechanism would provide few off-path predictions and thus the number of incorrectly predicted cycles would be

Parameter	Value
Architecture	Intel Sunny-Cove
Issue Width	6-way
Retirement Width	6-way
Branch Predictor	TAGE-SC-L 64K
BTB	8K entries
IBTB	2K entries
FTQ	32 entries
FTQ Max FTs per Cycle	2
ROB	352 entries
Reservation Station	125 entries
Data Prefetcher	Stream
Instruction Prefetcher	FDIP
Load Buffer	64 entries
Store Buffer	64 entries
L1-I Size	32KB 8-way
L1-I Latency	3 cycles
L1-D Size	48KB 12-way
L1-D Latency	4 cycles
L2 Size	512KB 8-way
L2 Latency	13 cycles
L3 Size	2MB 16-way
L3 Latency	36 cycles
Memory	DDR4-2400
Frequency	3.2 GHz

Table 4.2: Scarab simulation parameters. Parameters are chosen to emulate the Intel Sunny-Cove architecture.

relatively low. To counteract this, our accuracy is normalized by the number of off-path cycles, not total cycles. This is because the number of off-path cycles is roughly proportional to the number of mispredictions, so this cancels out any correlation of our mechanism with the total number of mispredictions. It is also more representative of the cycles we are attempting to predict.

Based on these insights, we define our adjusted accuracy as follows:

$$accuracy = 1 - \frac{early\_cycles + late\_cycles}{total\_offpath\_cycles} \tag{4.1}$$

With this definition, an accuracy of 1.0 would mean every cycle was correctly predicted. There is no lower bound on this accuracy; it can be negative. A negative adjusted accuracy means the predictor is performing worse than naively predicting all cycles on-path.

4.3 Performance and Power Approximation

Accurately simulating power in an architectural simulator is challenging. Power is dependent on the transistor layout of a design, and a single micro-architectural design can have any number of layouts that are logically equivalent but have different power characteristics. We approximate the power savings to be the percentage of cycles where our mechanism correctly predicts the processor off-path. In practice, this is higher than the actual number, as the backend won't be clock gated and clock gating only reduces dynamic power consumption.

Powering down the frontend at the incorrect time will decrease performance, as useful fetch targets and prefetches will not be produced. To approximate this, we consider the ratio

of cycles where the mechanism predicts off-path too early out of total cycles to be the percent slowdown incurred. The same inaccuracy applies to this approximation; not all parts of the processor will cease operation, so this approximation is higher than the actual slow-down would be.

4.4 I2P Performance

Figure 4.1 shows the adjusted accuracy of I2P with and without frontend throttling. On average, without throttling I2P has 41% accuracy for SPEC benchmarks and 59% accuracy for DB workloads. The mechanism has significantly worse accuracy when the frontend is throttled, with negative accuracy for both benchmark suites. The accuracy decrease from throttling the frontend comes from the penalty incurred by stalling too early. When the frontend is gated off-path, there will be minimal performance degradation, as all of the necessary fetch targets have already been pushed to the FTQ. However, if the frontend is gated too early, FTs that should be executed will not be pushed into the FTQ until the op that triggered the I2P off-path prediction is resolved. In the common case this takes between 50 and 200 cycles, and for many of these cycles the backend will be starved of useful instructions that it otherwise would have executed.

4.5 Comparison with Pipeline Gating

In this section, we compare our mechanism against the previous work, Pipeline Gating [32]. Figure 4.2 shows the accuracy of I2P vs pipeline gating, for SPEC and DC work-

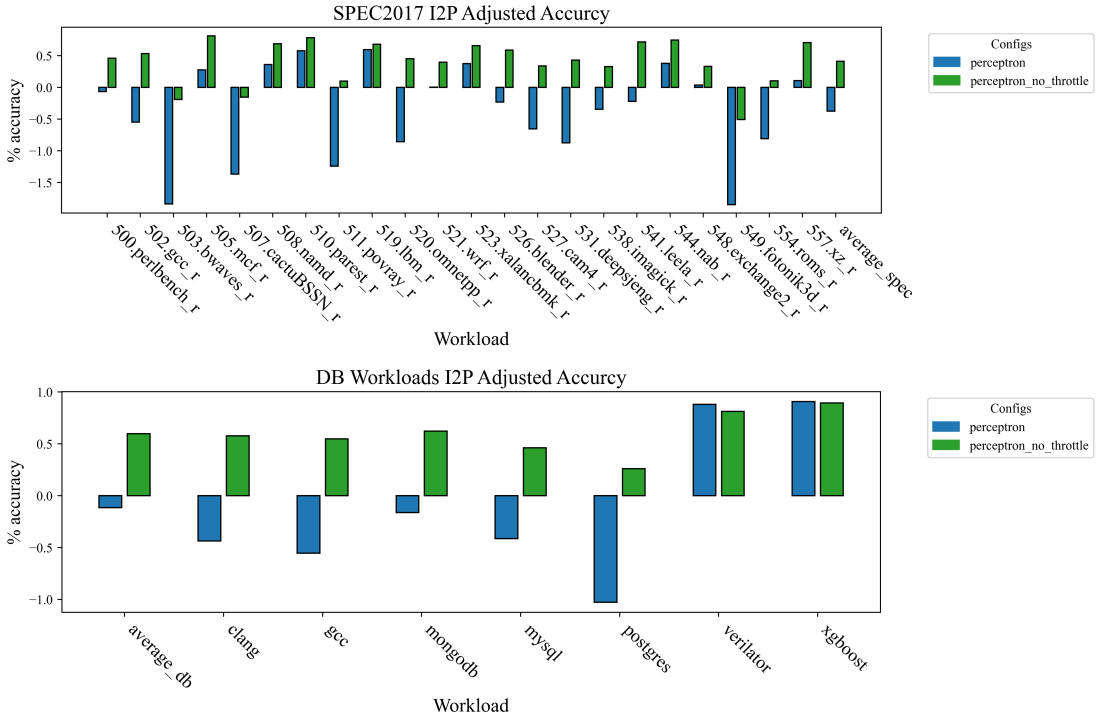

Figure 4.1: Adjusted accuracy for I2P Perceptron, with and without frontend throttling. Without frontend gating accuracy is positive and above 50% for many workloads. With frontend accuracy is negative for most workloads, indicating I2P is performing worse than predicting everything on-path.

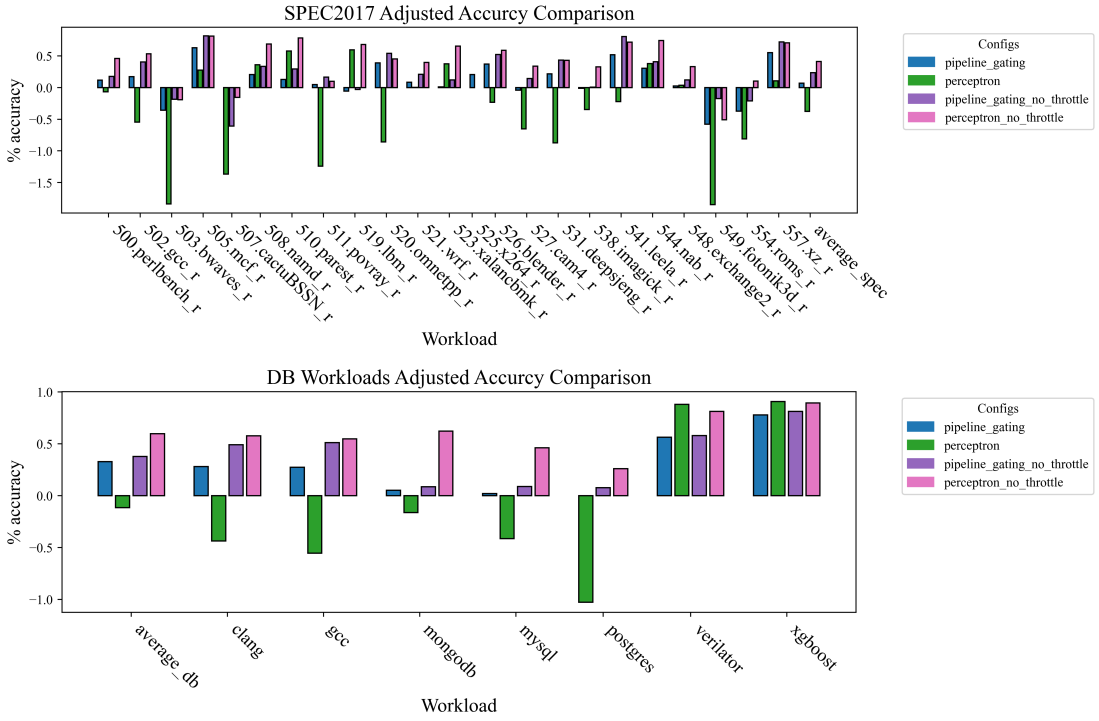

Figure 4.2: Comparison of I2P Perceptron against Pipeline Gating. The first two configurations, **pipeline_gating** and **perceptron** gate the frontend when they predict the frontend to be off-path. The second two configurations, **pipeline_gating_no_throttle** and **perceptron_no_throttle**, do not gate the frontend. The latter two configurations have no performance impact.

loads. Each mechanism was run with and without throttling the frontend on an off-path prediction. On average, without stalling the frontend I2P has better accuracy than Pipeline Gating for both SPEC and DC workloads. However, when stalling the frontend on an off-path prediction, Pipeline Gating has better accuracy for both SPEC and DC workloads.

Figure 4.3 shows the percent of cycles predicted off-path while the frontend was still on-path (green) and percent of cycles correctly predicted off-path (blue). For all workloads, the green bar for the I2P with frontend gating configuration is much larger than for all other configurations, meaning that I2P is spending a large amount of cycles incorrectly predicting the

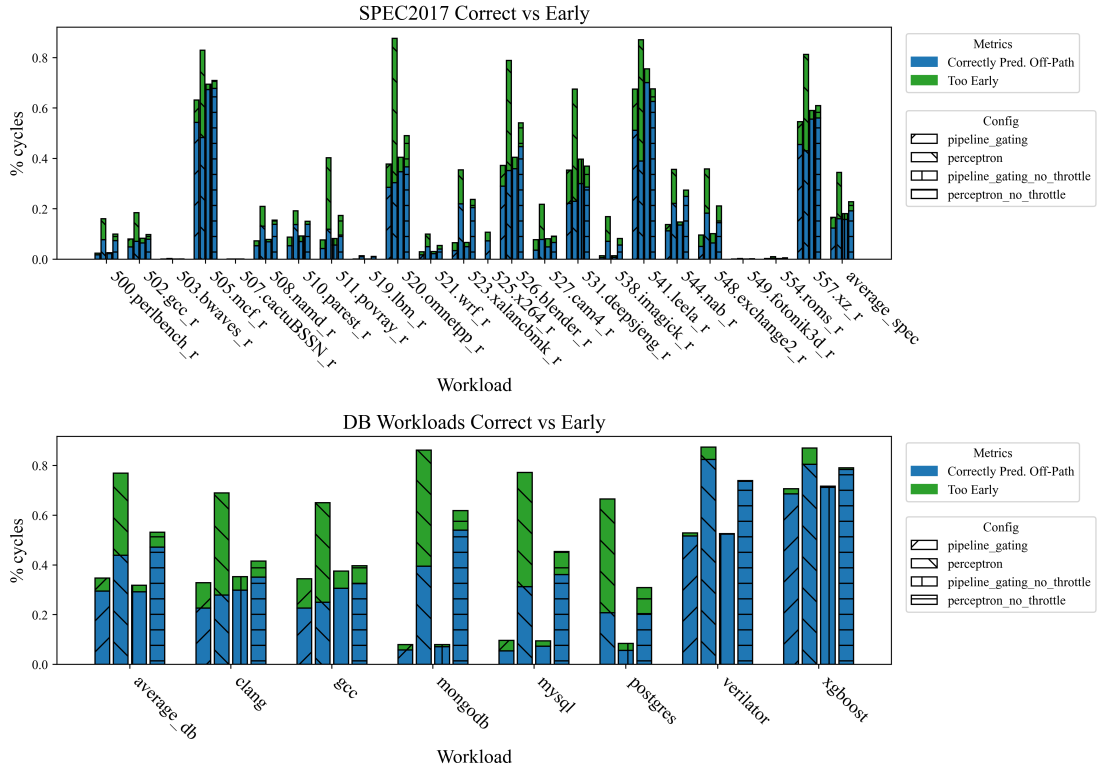

Figure 4.3: Percent of cycles predicted off-path correctly vs early. The green bar indicates the cycels where mechanism predicted off-path while the frontend was still on-path, while the blue bar indicates cycles where the mechanism predicted off-path while the frontend was actually off-path. For most workloads I2P perceptron performs better than Pipeline gating in predicting off-path cycles, but worse in early predictions.

processor is off-path. Notably, this bar is 4-5 times larger than the same bar for I2P without frontend gating for most workloads. Pipeline gating also sees an increase in cycles incorrectly predicted off-path with frontend gating, but the increase is much smaller than that of I2P.

This difference comes from the time spent gating the frontend once a mechanism has predicted off-path. Pipeline gating will stop gating the frontend when the number of low confidence branches in the pipeline drops below the threshold, which stops it from frontend gating for long periods of time. In contrast, I2P only stops frontend gating when the instruction

that triggered the off-path prediction resolves. Potential strategies for resolving this issue are detailed in the next section, Future Work.

Using the power and performance approximation outlined in section 4.3, it is clear that the performance impact of I2P is far too large to justify the power savings. While on average it predicts more off-path cycles correcly than pipeline gating, corresponding to higher power savings (Figure 4.3), slowdown is on average over 20% for SPEC2017 and 30% for the DC workloads. However, without stalling the frontend I2P is a better off-path confidence mechanism than Pipeline Gating and can be used for prefetch throttling. Analysis of I2P for prefetch throttling is beyond the scope of this work.

Chapter 5

Conclusion and Future Work

While I2P outperforms Pipeline Gating in terms of accuracy without frontend gating, it performs significantly worse when the frontend is throttled on an off-path prediction. Future work includes addressing the high impact performance impact of early off-path predictions. Possible methods for addressing this include:

- **Aggregating predictions as an off-path confidence.** When I2P is predicting on-path, instead of gating on the first off-path prediction, I2P can track consecutive off-path predictions and begin gating when this value crosses a threshold. The same is possible for stopping frontend gating; rather than waiting for the trigger instruction to resolve, I2P can continue predicting and return on-path when the number of consecutive on-path predictions crosses a threshold. The first case would prevent I2P from predicting off-path too early, and the second would prevent it from continuing to frontend gate after an early prediction.

- **Using the perceptron value as a confidence estimate.** While the final prediction is

a binary value, the perceptron algorithm produces a signed number. This number can be interpreted as a confidence estimate, where larger numbers are more likely to mean the frontend is off-path. Rather than predicting off-path when the sign of the number is positive, I2P can predict off-path when this number is greater than a confidence threshold. This method can be combined with the previous method.

- **Detecting when I2P is performing poorly.** Even with the above optimizations, I2P will likely perform worse on some workloads than others. Adding a small detection mechanism to track how often I2P is hurting performance could reduce negative impacts on workloads with poor I2P accuracy.

Another way to increase accuracy could be to refine the features. The simplified version of the Perceptron algorithm commonly used in hardware eliminates costly multiplication by updating weights with either -1 or 1, not learning rate * feature value. This works well for binary features, such as whether a branch in the history was taken or not taken, but doesn't work well for continuous values, e.g. the number of low-confidence branches in the pipeline. Empirically determining thresholds to convert continuous values, for example FT length and BP confidence, to binary values or smaller ranges could increase the efficiency of the simplified Perceptron training algorithm.

The final future direction for this work is to use this mechanism for reducing instruction cache pollution. Multiple recent instruction prefetchers attempt to take into account whether the frontend is on-path. Combining their methods with I2P could improve the efficacy of their mechanisms.

Bibliography

[1] Samira Mirbagher Ajorpaz, Elba Garza, Sangam Jindal, and Daniel A Jiménez. Exploring predictive replacement policies for instruction cache and branch target buffer. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 519–532. IEEE, 2018.

[2] D.N. Armstrong, Hyesoon Kim, O. Mutlu, and Y.N. Patt. Wrong Path Events: Exploiting Unusual and Illegal Program Behavior for Early Misprediction Detection and Recovery. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 119–128, 2004.

[3] D.N. Armstrong, Hyesoon Kim, O. Mutlu, and Y.N. Patt. Wrong path events: Exploiting unusual and illegal program behavior for early misprediction detection and recovery. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 119–128, 2004.

[4] Grant Ayers, Nayana Prasad Nagendra, David I August, Hyoun Kyu Cho, Svilen Kanev, Christos Kozyrakis, Trivikram Krishnamurthy, Heiner Litz, Tipp Moseley, and Parthasarathy Ranganathan. Asmdb: understanding and mitigating front-end stalls in warehouse-scale computers. In *ISCA-46*, 2019.

[5] James Bonanno, Adam Collura, Daniel Lipetz, Ulrich Mayer, Brian Prasky, and Anthony Saporito. Two level bulk preload branch prediction. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 71–82. IEEE, 2013.

[6] Brian K. Bray and M. J. Flynn. Strategies for branch target buffers. In *Proceedings of the 24th annual international symposium on Microarchitecture - MICRO 24*, pages 42–50, Albuquerque, New Mexico, Puerto Rico, 1991. ACM Press.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[8] Ioana Burcea and Andreas Moshovos. Phantom-btb: a virtualized branch target buffer design. *Acm Sigplan Notices*, 44(3):313–324, 2009.

[9] Ioana Burcea and Andreas Moshovos. Phantom-btb: a virtualized branch target buffer design. 37(1):313–324, March 2009.

[10] Chen-Yong Cher and TN Vijaykumar. Skipper: a microarchitecture for exploiting control-flow independence. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pages 4–15. IEEE, 2001.

[11] Barry Fagin. Partial resolution in branch target buffers. *IEEE Transactions on Computers*, 46(10):1142–1145, 1997.

[12] Hongliang Gao, Yi Ma, Martin Dimitrov, and Huiyang Zhou. Address-branch correlation: A novel locality for long-latency hard-to-predict branches. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 74–85, 2008.

[13] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. Evolution of the samsung exynos cpu microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 40–51, 2020.

[14] Saurabh Gupta, Niranjan Soundararajan, Ragavendra Natarajan, and Sreenivas Subramoney. Opportunistic early pipeline re-steering for data-dependent branches. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 305–316, New York, NY, USA, 2020. Association for Computing Machinery.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017.

[16] Andrew D Hilton and Amir Roth. Ginger: Control independence using tag rewriting. *ACM SIGARCH Computer Architecture News*, 35(2):436–447, 2007.

[17] Daniel A. Jiménez. Multiperspective perceptron predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, June 2016.

[18] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture (HPCA'01), Nuevo Leone, Mexico, January 20-24, 2001*, pages 197–206. IEEE Computer Society, 2001.

[19] Daniel A Jiménez and Elvira Teran. Multiperspective reuse prediction. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 436–448. IEEE, 2017.

[20] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd ISCA*, 2015.

[21] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. Twig: Profile-guided btb prefetching for data center applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 816–829, 2021.

[22] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci. Whisper: Profile-guided branch misprediction elimination for data center applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 19–34. IEEE, 2022.

[23] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn. Virtual program counter (vpc) prediction: Very low cost indirect branch prediction using conditional branch prediction hardware. *IEEE Transactions on Computers*, 58(9):1153–1170, 2009.

[24] Ryotaro Kobayashi, Yuji Yamada, Hideki Ando, and Toshio Shimada. A cost-effective branch target buffer with a two-level table organization. In *Proceedings of the 2nd International Symposium of Low-Power and High-Speed Chips (COOL Chips II)*, 1999.

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

[26] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 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*, ASPLOS '18, page 30–42, New York, NY, USA, 2018. Association for Computing Machinery.

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

[28] Lee and Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[29] Chit-Kwan Lin and Stephen J. Tarsa. Branch prediction is not a solved problem: Measurements, opportunities, and future directions. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 228–238, 2019.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[31] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pages 132–141, 1998.

[32] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pages 132–141, 1998.

[33] Scott McFarling. Combining branch predictors. Technical report, Citeseer, 1993.

[34] Pierre Michaud. An alternative tage-like conditional branch predictor. *ACM Trans. Archit. Code Optim.*, 15(3), August 2018.

[35] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. UDP: Utility-Driven Fetch Directed Instruction Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1188–1201, June 2024.

[36] Reena Panda, Paul V Gratz, and Daniel A Jiménez. B-fetch: Branch prediction directed prefetching for in-order processors. *IEEE Computer Architecture Letters*, 11(2):41–44, 2011.

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

[38] Stephen Pruett and Yale Patt. Branch runahead: An alternative to branch prediction for impossible to predict branches. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 804–815, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Glenn Reinman, Todd Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. *ACM SIGARCH Computer Architecture News*, 27(2):234–245, 1999.

[40] Glenn Reinman, Brad Calder, and Todd Austin. Fetch directed instruction prefetching. In *MICRO-32*. IEEE, 1999.

[41] R. Sendag, A. Yilmazer, J.J. Yi, and A.K. Uht. Quantifying and reducing the effects of wrong-path memory references in cache-coherent multiprocessor systems. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10 pp.–, 2006.

[42] Resit Sendag, Joshua Yi, and Peng-fei Chuang. Branch Misprediction Prediction: Complementary Branch Predictors. *IEEE Computer Architecture Letters*, 6(2):49–52, 2007.

[43] Resit Sendag, Joshua Yi, and Peng-fei Chuang. Branch misprediction prediction: Complementary branch predictors. *IEEE Computer Architecture Letters*, 6(2):49–52, 2007.

[44] André Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

[45] André Seznec. Exploring branch predictability limits with the mtage+ sc predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, page 4, 2016.

[46] André Seznec. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, Seoul, South Korea, June 2016.

[47] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *J. Instr. Level Parallelism*, 8, 2006.

[48] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[49] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. Pdede: Partitioned, deduplicated, delta branch target buffer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 779–791, 2021.

[50] Akash Sridhar, Anujan Varma, and Nicholas Brummell. *Load Driven Branch Predictor (LDBP)*. PhD thesis, 2021. AAI28317175.

[51] David Tarjan and Kevin Skadron. Merging path and gshare indexing in perceptron branch prediction. *ACM Trans. Archit. Code Optim.*, 2(3):280–300, September 2005.

[52] Stephen J Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, and Hong Wang. Improving branch prediction by modeling global history with convolutional neural networks. *arXiv preprint arXiv:1906.09889*, 2019.

[53] Santhosh Verma, Benjamin Maderazo, and David M. Koppelman. Spotlight - a low complexity highly accurate profile-based branch predictor. In *2009 IEEE 28th International Performance Computing and Communications Conference*, pages 239–247, 2009.

[54] S. Vlaovic, E.S. Davidson, and G.S. Tyson. Improving BTB performance in the presence of DLLs. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 77–86, December 2000. ISSN: 1072-4451.

[55] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[56] WikiChip. Sunny cove - microarchitectures - intel. `https://en.wikichip.org/wiki/intel/microarchitectures/sunny_cove`.

[57] Tse-Yu Yeh and Yale N Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture*, pages 51–61, 1991.

[58] Cliff Young, Nicolas Gloy, and Michael D Smith. A comparative analysis of schemes for correlated branch prediction. *ACM SIGARCH Computer Architecture News*, 23(2):276–286, 1995.

[59] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *MICRO-53*. IEEE, 2020.