The Dissertation Committee for Lingzhe Cai
certifies that this is the approved version of the following dissertation:

Branch Prediction without the Full View of Global History

Committee:

Yale N. Patt, Supervisor

Poulami Das

Mattan Erez

Christopher J. Rossbach

Dam Sunwoo

Branch Prediction without the Full View of Global History

by
Lingzhe Cai

Dissertation

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

The University of Texas at Austin
May 2026

Acknowledgments

This dissertation would not have been possible without the help and support of my mentors and friends. Although it will not be possible to name each of them, I will try my best.

First and foremost, I would like to thank my advisor, Dr. Yale Patt. When I started my PhD, I was a 23 year old with no research experience. Not only did you teach me about what it means to do good research, you showed me the importance of finding a good research topic and the importance of good communication. You always encouraged us to take on big challenges, and you have created an environment in our group where truly impactful work is expected out of everyone. These are qualities that have greatly improved my thesis, and I will hold on to them for the rest of my life.

My fellow HPS group members have provided me with a environment where I can always ask anyone for help and they would always be willing to help. They have challenged me to produce the best work. Specifically, I would like to thank

- Faruk Guvenilir for his help during my very first research project on Spectre and Meltdown, and for his advice on looking for job in the industry.
- Ben Lin for his humor and knowledge of the memory system.
- Stephen for teaching me everything from branch prediction to Texas BBQ. I appreciate you for welcoming me to Austin on my visit day. Your deep knowledge of the CPU has always been a role model for me. This thesis is largely inspired by some of the conversations we had when I was looking for a topic.
- Siavash Zangeneh for teaching me everything about TAGE and C++. Your deep understanding of branch prediction has been instrumental in my research.

Most importantly, you introduced me to climbing, which is now one of my favorite activities.

- Aniket Deshmukh for being my partner in crime for everything in the past 8 years from research to life. Your creativity and encouragement have positively influenced my PhD experience. You are always willing to explore even the craziest research ideas with me, and I will forever cherish those impromptu meeting with you and a whiteboard. I could not imagine what my PhD experience would be like if you were not my partner.
- Rob Chappell for providing me with invaluable career advice.
- Jose Joao and Rustam Miftakhutdinov for helping me navigating through my internships.
- Leticia Lira for keeping our group together.

I thank Dam Sunwoo, Krishnendra Nathella, Jonathon Combs, Vikash Agarwal, Chris Browne, Kulin Kothari, Muawya Al-Otoom, Ilhyun Kim for the internships I have had during my PhD. These internships greatly helped me understand the state-of-the-art in CPU design and what the real challenges are.

I thank Mattan Erez, Poulami Das, Christopher Rossbach and Dam Sunwoo for serving on my dissertation committee. Your feedback has greatly improved the quality of this dissertation. I would also like to thank Derek Chiou. Even though you did not serve on my committee, your advice over the years on graduate school and research in general has help me immensely. Your generosity towards others have inspired me to be a better person.

I would like to thank Professor Daniel Chang for introducing me to computer architecture at Rose-Hulman Institute of Technology. Computer architecture quickly became my favorite class because of your humor and great teaching.

I would also like to thank all my friends for keeping me from going insane during this period of my life. Specifically, I would like to thank:

- Prateek Sahu, Pranav Kumar and Willy Vasquez for helping me transition into graduate school and all the late night activities during my first year.
- Jeageun Jung, Jaeyoung Park, Kayvan Mansoorshahi, Wenqi Yin, Anyesha Ghosh, and Matthew Barondeau for being a great friend and always easy to talk to.
- Alex Hsu for helping me realize that computer architecture is more than just CPUs.
- Ali Fakhrzadehgan for always willing to provide detailed feedback on research ideas, and for helping me with my security project when I did not understand anything about security.
- Sophia Jiang for always putting others before herself, and for helping me throughout my Optiver interview process.
- Evan Lai for being a great team member during the CBP competition.
- Sean Stephens and Abbie Dowd for being my partner in microarchitecture. Both of you have been instrumental in us finishing the course project.
- Colin, Sidra, Amelia Berg for flying down to Austin for my defense and your friendship over the past decade.
- Josh Palamutam, Aaron Prins and Alessandro Adarve for their friendship and always picking up the paycheck when going out to eat. Especially, I thank Josh for forcing me to use AI for coding when I was too stubborn to give it a try otherwise.
- Chenyang Wu for putting up with my craziness for the past 4 years.

I would also like to thank the Barnett family for going above and beyond for me whenever I needed anything. When I first arrived in the US, you welcomed me with open arms and made me feel like home. I have always enjoyed visiting during Thanksgiving and I hope to continue this tradition.

Lastly, I would like to thank my parents, Huaqin Cai and Li Lin. You have taught me the importance of being kind to others, allowed me to make mistakes, gave me the opportunities to correct my mistakes, and most importantly provided me with the safety net when I was not able to fix my own mistakes. While I have not spent much time home since moving to the US, your support for me both emotionally and financially have never stopped. Much of this dissertation would never have happened is it wasn't for the value you instill upon me, and for that, I am truly grateful.

-Lingzhe Cai, April 2026, Austin, Texas

Abstract

Branch Prediction without the Full View of Global History

Lingzhe Cai, PhD
The University of Texas at Austin, 2026

SUPERVISOR: Yale N. Patt

High-performance CPUs depend on accurate branch prediction. Decades of research results in complex prediction algorithms that draw correlations from deep in the control flow history to accurately predict the branch. However, implementing such complex designs faces two key challenges: prediction latency and history register length. This dissertation leverages the observation that most branches are highly predictable and contribute little new control-flow information. Using this insight, I first propose an energy-efficient ahead predictor, reducing the high energy costs of prior designs while hiding the predictor latency. Secondly, I show that excluding predictable branches from the global history not only does not hurt prediction accuracy, but can improve prediction accuracy by effectively extending usable history length, which improves the prediction accuracy. Together, these techniques help unlock the true potential of prior research on branch prediction, offering new directions for future branch predictor design.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Single-thread performance remains a critical metric in modern processor design as many important workloads continue to depend on it. Out-of-Order processors based on the HPS model[43, 44] have achieved wide-spread success in commercial products. These processors achieve their performance through a sophisticated out-of-order engine to dynamically identify independent instructions that can be executed at the same time, paired with complex predictors to enable processors to speculatively fetch instructions beyond control-flow boundaries, maintaining a steady supply of correct-path instructions to the backend. As commercial products continue to increase both pipeline width and the out-of-order window to extract more instruction level parallelism and memory level parallelism to improve single thread performance, more speculative instructions are in-flight at any given time, meaning a misprediction forces a greater number of incorrectly fetched instructions to be squashed and the pipeline to be refilled from the correct path. This increased misprediction penalty makes accurate branch prediction even more important in the design of modern processors.

Modern branch predictors make accurate predictions by learning the correlation between the control flow leading up to a branch and the direction of the branch. This concept was first introduced by the 2-level predictor [65], and nearly all high-accuracy predictors [21, 26, 29, 31, 54, 56, 59, 63, 66] today follow the same key principle. The control flow is typically represented by a history of recently executed branches, stored in a global history register, which encodes the dynamic control flow leading up to a branch.

To capture this control flow context, state-of-the-art branch predictors update the history register on every branch, implicitly assuming that every branch outcome contributes new information about the program's control flow.[1] Under this assump-

[1]Industry predictors often adopt taken-only history[57], recording only the PC and targets of the

tion, the predictor must use the full global history for prediction as the direction of each branch potentially changes the context seen by future branches.

In this dissertation, I will show that this assumption is not only incorrect, but also gives rise to two important inefficiencies in branch prediction: prediction latency and history aliasing, both of which are discussed in detail in the following sections.

1.1 Predictor Latency

Modern predictor designs use large lookup tables and complex logic to generate a prediction, resulting in a multi-cycle latency to produce a prediction. Branch predictors use the full control flow history as input, meaning the history register must be updated with the outcome of the current prediction before the next prediction can begin. This creates a strict sequential dependency between predictions that prevents the predictor from being pipelined, and its latency cannot be hidden. Prior work has proposed two ways to solve this problem, but both have shortcomings.

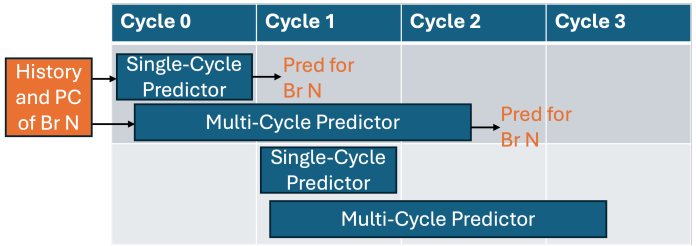

Figure 1.1: Multi-level Prediction Timing Diagram When Fast and Slow Predictors Agree

Multi-level prediction schemes [22] use a fast and simple predictor to provide an initial prediction, followed by one or more overriding predictors. The simple predictor produces a prediction in a single cycle. This allows the next prediction to start immediately on the next cycle using the results of the first prediction. The

taken branches. This is broadly information-equivalent, as the positions of not-taken branches are implicitly conveyed by the position of the taken branches.

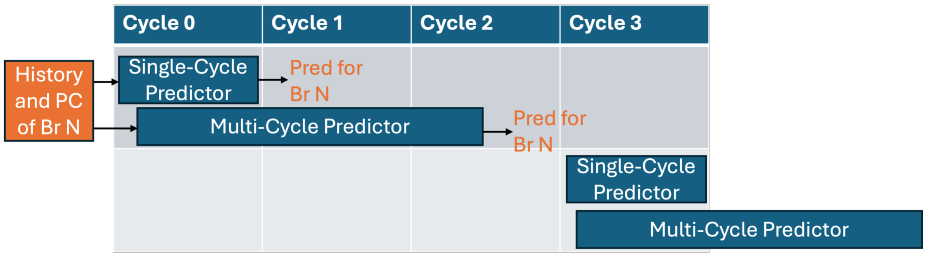

Figure 1.2: Multi-level Prediction Timing Diagram When Fast and Slow Predictors Do Not Agree

overriding predictor(s) takes multiple cycles to generate a more accurate prediction, either agreeing with or disagreeing with the prediction made by the single cycle predictor. Figure 1.1 shows the timing diagram when the fast and slow predictors agree with each other. In this case, the next prediction is started on the next cycle and no pipeline stalls are observed. But when the slower overriding predictor disagrees with the simple predictor, an early pipeline flush is issued, as shown in Figure 1.2. In this case, 2 bubbles appear in the pipeline. To quantify the effect of predictor latency on processor performance, I conducted an experiment across all SPEC2017 CPU benchmarks, showing that frequent disagreements between the two predictors can introduce significant performance degradation. In fact, eliminating predictor latency entirely in such designs can improve IPC by up to 6.48% in an aggressive out-of-order core. While this scheme is widely adopted in commercial designs, it still does not fully solve the prediction latency problem.

Ahead prediction [18, 20, 33, 36, 37, 53, 60] breaks the sequential dependency by predicting a future branch using the currently available history and PC information, allowing each prediction to begin earlier and hide multi-cycle latency. As shown in Figure 1.3, the prediction for BR2 is initiated using BR0's history and PC, rather than waiting for BR1's prediction to complete and update the history first.

However, this means the directions of BR0 and BR1, the intermediate branches between when the prediction for BR2 is initiated and when it is needed, are ab-

sent from the history used for predicting BR2, reducing prediction accuracy. Prior work [23, 52, 55] observed that by the time BR2's prediction is actually needed, all intermediate branches have already been predicted. By generating a prediction for each possible missing history pattern, the predictor can simply select the correct one once the intermediate outcomes are known.

However, prior work still assumes that the full history is needed for prediction, meaning any one of the missing history patterns could materialize and affect the outcome. Under this assumption, a separate prediction must be generated for every possible combination of missing history bits, one entry read per prediction table per pattern. Since the number of missing history patterns grows exponentially with ahead distance, the number of table reads and associated energy also increase exponentially. For example, prior work [55] increased the number of bits read per prediction by 32× when predicting five branches ahead, resulting in a 14.6× increase in per-prediction energy. As directional branch prediction accounts for approximately 3–4% of total core power [8, 38, 42], this overhead is substantial, making prior approaches to ahead prediction impractical.

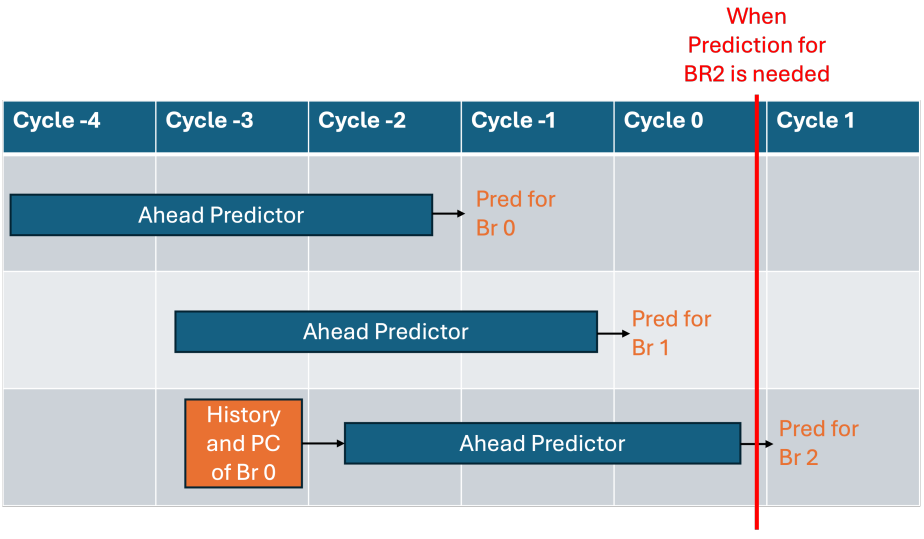

Figure 1.3: Ahead Prediction Timing Diagram

1.2 History Length

Accurate predictors extract correlations that may span thousands of dynamic branches. For example, TAGE-SC-L [56], the winner of CBP5, uses a 3000-bit global history register. RUNLTS [31], the winner of CBP6, uses a 4316-bit history register[2]. This demonstrates that correlations from deep in the history can be beneficial for prediction accuracy. In contrast, commercial designs [57, 64] typically track fewer than 100 taken branches. My experiments show that extending the size of the history register from 256 bits to 3000 bits reduces the misprediction rate by 0.4 MPKI using TAGE [56] over the 105 traces provided by CBP6[5].

Although increasing the size of the global history register can be achieved with a longer register, the real cost is the need to recover the correct history after pipeline flushes caused by a misprediction. Recovering the correct history requires checkpointing the history for each branch in the processor. Since modern processors can have very large instruction windows, the total checkpoint storage can be substantial. For example, an instruction window of 1000 would require 250 checkpoints[3], which means increasing the history size can drastically increase the size of the checkpoints.

Increasing the size of the checkpoint also increases the time needed to reconstruct the correct history after a branch misprediction is detected. To hold these larger checkpoints, larger storage tables are needed, which results in a multi-cycle access latency. This multi-cycle access latency will delay the pipeline restart because the predictors cannot start until the history is recovered. The amount of time needed to restart the pipeline after a branch misprediction directly impacts performance; prior work [12] has shown that each additional cycle in restarting the pipeline after a flush incurs around a 1% performance loss. Both the storage overhead of the

[2]The size of the history register and its checkpoints are not part of storage overhead in the championship branch prediction competitions, therefore participants can use an impractically large history size as long as it can provide benefits

[3]assuming 1 in every 4 instructions are branches.

checkpoints and the latency to read the checkpoints prevent a long history register implementation. While alternative methods for recovery like backwards-shifting and circular-buffer implementations exist, they come at high energy, timing, or area costs [51].

Although predictor latency and history length are typically studied as separate problems, both originate from the same flawed assumption in modern branch predictor design: that every branch outcome contributes new information to the control flow. In the context of predictor latency, this assumption forces ahead predictors to consider all 2^N possible missing history patterns when predicting N branches ahead as it assumes each intermediate branch provides new information, making the energy cost grow exponentially with ahead distance. In the context of history length, the same assumption implies that every branch outcome must be recorded in the history register, making the cost of storing and recovering long histories grow linearly with history length. In both cases, the assumption is unnecessarily conservative. As I will show later, predictable branches, which are the majority of the dynamic branches in a program, do not add new information to the control flow, leaving ample room for optimization.

1.3 Predictable Branches

In my thesis, I challenge the traditional wisdom that every branch introduces an extra path in the control flow. I will show that predictable branches do not lead to a new path in the control flow, thus providing no additional information to help identify the control flow.

Branch prediction algorithms exploit the correlation between control flow (history) and branch direction [9, 14]. These correlations arise from the program structure, data dependencies, or inherent input regularities. History-based branch predictors learn the correlation between the control flow leading up to the branch and the direction of the branch through either a neural network [21, 26, 66], or through lookup

tables [31, 35, 54, 56, 59, 63, 65] that track the majority direction of the branch under each control flow.

For a branch to be predictable, regardless of the mechanism used, it must exhibit stable behavior—nearly always taken or nearly always not taken—under that control flow. Conversely, a branch is unpredictable if its behavior does not converge to a stable majority direction even under the longest supported history. Importantly, predictability is history-dependent: a branch may be predictable under some control flows but not others.

Modern history based branch predictors such as TAGE-SC-L can achieve a very high prediction accuracy (e.g., 97.06% across the 105 traces in the 6th Championship Branch Prediction Competition), suggesting that most branches in a program are predictable based on the control flow leading up to the branch.

1.3.1 Key Insight

```
//A, B are random variables

If (A) { } //BR 1
...
If (B) { } //BR 2
...
If (A & B){ } // BR 3
```

Figure 1.4: Code Example of a Predictable Branch

Conventional wisdom suggests that each branch can lead to two control flows because it can be either taken or not taken. Under this view, three branches would yield eight distinct control flows. However, this exponential growth is illusory when branches are predictable.

In Example 1.4, there are three if blocks. Each if block is translated into a branch instruction during code generation, and the branch is taken if its condition

evaluates to true, skipping the instructions inside the block. First, notice that branch 3 is highly predictable: its direction can be determined from the outcomes of branch 1 and branch 2, making the branch completely predictable. For instance, a 4-entry pattern history table indexed by the directions of branch 1 and branch 2 can achieve 100% prediction accuracy. Second, there are only 4 control flow paths for the 3 branches in the example, as shown in Table 1.1. Although branch 3 can be taken or not taken at runtime, its direction is completely dependent on Branch 1 and Branch 2. As a result, it does not introduce an additional control-flow path beyond those already established by branches 1 and 2, which explains why the total number of control flows after the three if blocks is only four instead of eight. Notice here that the reason it does not introduce a new control flow is exactly what makes this branch predictable.

	Branch1	Branch2	Branch3
Path 1:	Taken	Taken	Taken
Path 2:	Not Taken	Taken	Taken
Path 3:	Taken	Not Taken	Taken
Path 4:	Not Taken	Not Taken	Not Taken

Table 1.1: Control-flow Outcomes of the Three Branches Example

More generally, for any branch whose direction is stable under a given control flow, it does not introduce another path. This property is exactly what makes a branch predictable. This insight—that predictable branches do not create new control-flow paths—forms the foundation for the rest of this dissertation.

If predictable branches dominate execution, then the conventional assumption that every branch contributes new control-flow information can lead to significant inefficiency in building branch predictors. The next section empirically measures the percentage of branches that are predictable at run time across the 105 traces provided by cbp6 [5].

1.3.2 Experimental Results

I performed an experiment to determine the number of dynamic branches that are predictable. I define predictability as follows: given a particular history and PC, what percentage of outcomes correspond to the majority direction? For example, if under a particular control flow, a branch is taken 999 times and not taken once, then the branch is 99.9% predictable under this control flow.

	No History		32-bit History		64-bit History	
Predictability	**Fraction**	**Acc**	**Fraction**	**Acc**	**Fraction**	**Acc**
≥99.9%	46.92%	99.76%	74.90%	99.04%	84.85%	98.04%
≥99.0%	55.57%	99.75%	80.67%	99.09%	89.28%	98.12%
≥97.0%	62.12%	99.67%	85.46%	99.05%	92.32%	98.13%
≥95.0%	66.15%	99.57%	88.12%	99.00%	93.92%	98.11%
<95%	33.85%	92.00%	11.88%	82.27%	6.08%	80.01%

Table 1.2: Branch predictability and prediction accuracy under different history lengths.

For each retired conditional branch during program execution, I record the PC, the global history, and the branch outcome. At the end of program execution, I compute the predictability for each unique (PC, history) pair. The experiment is conducted over the 105 traces provided by CBP6 [5]. Table 1.2 reports the distribution of predictability and prediction accuracy[4] under no history, 32-bit history, and 64-bit history. The results confirm that my predictability metric aligns with the empirical results from the predictor: branches that are highly biased toward one direction under a given history are predicted accurately, while branches that are not at least 95% biased toward a direction see a sharp drop in prediction accuracy. Two other key observations emerge. First, longer history significantly increases the fraction of predictable branches. Second, the vast majority of dynamic conditional branches (84.85%) are more than 99.9% predictable under 64-bit history, and those branches are

[4]using a 64KB TAGE-SC-L

predicted correctly 98.04% of the time, confirming that high predictability translates directly to high prediction accuracy in practice.

1.3.3 Applying this Insight

In this dissertation, I apply the insight that predictable branches do not create new control-flow paths to improve the branch prediction in the following two ways:

Enabling Ahead Prediction with Practical Energy Constraints: Because skipping over a predictable branch does not introduce an extra pattern and the majority of branches at runtime are very predictable, most of the 2^N missing history patterns do not materialize when predicting N branches ahead. Previous work considers all 2^N missing history patterns when predicting N branches ahead, unnecessarily increasing the predictor energy when all patterns are considered for prediction. An ahead predictor considering only the missing history patterns that materialize at run time (usually only 1 or 2 when predicting 5 branches ahead) can drastically decrease the per-prediction energy associated with ahead prediction (from 14.6× to 1.5×).

Achieving a Longer Effective History via Pruning Predictable Branches: Because predictable branches do not contribute additional control-flow information under a given history, recording their outcomes in the global history register is unnecessary. By selectively omitting these updates, the predictor preserves all relevant information while effectively extending the usable history length without increasing storage.

1.4 My Contributions

The contributions of this dissertation are:

- I show that the number of missing history patterns observed at runtime is far smaller than the theoretical maximum.

- I propose an efficient ahead predictor that explicitly tags counters with missing-history patterns, allowing per-prediction energy to scale linearly with ahead distance.
- I design a branch history pruning algorithm by skipping the history updates from predictable branches, increasing the effective length of the history register.
- I design a dynamic mechanism to identify predictable branches and selectively skip their history updates, effectively extending usable history length, improving prediction accuracy.

1.5 Thesis Statement

Predictable branches do not introduce new control-flow information, and exploiting this property enables more efficient branch predictors: in the context of ahead prediction, the scarcity of history patterns that materialize at runtime enables practical energy-efficient ahead prediction; and in the context of history length, omitting predictable branches from history updates extends the effective history length without increasing storage or recovery overhead.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 addresses the problem of predictor latency — the challenge of delivering branch predictions fast enough to keep up with modern wide-issue processors. It surveys prior approaches, analyzes the number of missing history patterns observed in practice, and presents the design of an efficient ahead predictor along with its integration with the processor core and a full evaluation. Chapter 3 addresses the problem of history length — the challenge of capturing long-range branch correlations without incurring prohibitive storage, latency, and energy costs. It presents the key insight that predictable branches are redundant in the history register, analyzes what should and should not

be skipped, describes the full implementation including the locked table and blacklist mechanisms, and evaluates the resulting predictor against state-of-the-art designs. Chapter 4 concludes the dissertation with a summary of contributions and a discussion of directions for future work.

Chapter 2: Predictor Latency

2.1 The Problem

Branch predictors use lookup tables to identify the correlation between the control flow leading up to a branch and its direction. As branches in the program see many control flow patterns, large tables are necessary to capture all of them and provide high-accuracy predictions. TAGE-SC-L [56] is the winner of Championship Branch Prediction 5 (CBP5) [1] competition, where a 64KB version achieves 25.3% fewer mispredictions compared to an 8KB TAGE-SC-L. In terms of logic complexity, the initial two-level predictor design [65] with a global history and a global pattern history table only required a single table lookup, but modern predictors like TAGE require hashing, table look-ups, and a complex selection function to generate the final prediction. Perceptron-based predictors [21, 26] require table look-ups and a dot product computation for the final prediction. A recent proposal [66] even uses a small on-chip CNN inference engine to generate predictions. Both large storage and complex logic provide higher accuracy but increase predictor latencies.

To reduce the impact of this latency problem while maintaining high accuracy, industry uses multi-level branch prediction [2, 17, 22, 45]. A small predictor generates a prediction within the first cycle, but a larger and more accurate predictor may override that prediction in the next few cycles. While this design provides the high accuracy of a large predictor, it only provides the low latency of a small predictor when it agrees with the overriding predictor. Each disagreement between the two effectively stalls the prediction pipeline and adversely affects performance.

Fig. 2.1 shows the impact of overriding predictor latency on processor performance in SPEC CPU2017 [4] benchmarks. I use a 1K-entry table consisting of PC-tagged 2-bit saturating counters as the single cycle predictor and TAGE as the main predictor (the size of TAGE is fixed for all latencies in this study). A prediction

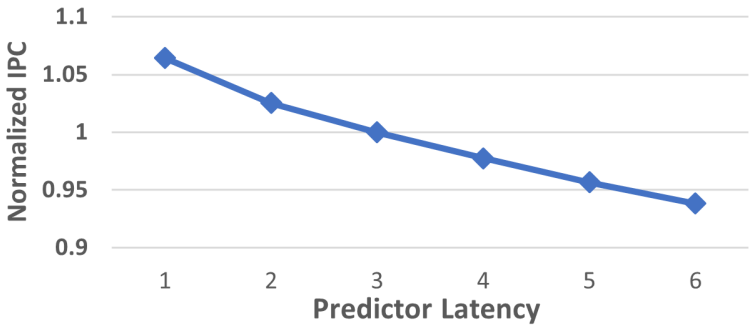

Figure 2.1: Performance Impact of BP Latency

packet is generated every cycle unless there is a disagreement between TAGE and the single cycle predictor. The IPC is normalized to a baseline with a 3-cycle overriding predictor. A latency of 1 means that TAGE's prediction is available at the end of the first cycle and used as the single-cycle predictor. Completely removing the predictor latency can provide a 6.48% IPC improvement. Each additional cycle of predictor latency decreases the overall performance by 2.5%.

2.2 Prior Work and Alternative Solution

2.2.1 Multi-Level Prediction

Pipelining predictors is challenging because the prediction of the current branch depends on the prediction of the preceding branch. Predictors use the current branch's PC and history as inputs, which are only available after the previous prediction is generated. Industry products [2, 17, 22, 45] solve this problem with a multi-level prediction scheme. A simple single-cycle predictor allows the next prediction to start on the next cycle. This is supported by an overriding predictor that is larger and has a longer latency but is more accurate. Both predictors start in the same cycle, but the overriding predictor's result arrives a few cycles later. This result is compared against the single-cycle prediction. On a disagreement, it overrides the single-cycle prediction via an early flush. Each early flush effectively stalls the prediction pipeline for N-1 cycles, where N is the latency of the overriding predictor. Single-cycle predictors are

usually extremely simple due to timing constraints and, therefore, are significantly less accurate than the overriding predictor. My experiments show that a 1K-entry PC-tagged 2-bit counter has 10.8x more mispredictions across the SPEC CPU2017 benchmarks compared to a 64KB TAGE-SC-L on average and can have 1000x more mispredictions in the worst case. Frequent early flushes significantly limit the prediction throughput and hurt performance.

Multi-level prediction schemes also limit scaling both in terms of storage and throughput. **First**, the storage budget of the overriding predictor is hard to scale as it increases predictor latency. The performance degradation from the increased latency often outweighs the performance improvement from having a larger predictor. Specifically, doubling the predictor size to 128KB leads to a 0.07 Misses-Per-Kilo-Instruction (MPKI) reduction across all of SPEC benchmarks but decreases IPC by 1.4%, assuming doubling the predictor size increases predictor latency by 1. Even for the two benchmarks most sensitive to predictor capacity, gcc and leela (with an MPKI reduction of 0.21 and 0.50, respectively), performance decreases by 0.1% for gcc and only increases by 0.9% for leela. **Second**, a multi-level predictor scheme limits the predictor throughput because a longer predictor latency increases the number of cycles during which the predictor is stalled. This decreases the effectiveness of a wider frontend, which is critical for high performance.

2.2.2 Ahead Prediction

Ahead prediction breaks the dependency between consecutive branches by using the current PC and history to predict a future branch. This technique hides the prediction latency, as the prediction of the future branch is not needed until a few cycles later. The number of branches skipped is called the ahead distance. However, ahead prediction hurts prediction accuracy as the same ahead history and PC could lead to multiple branches,[1] making it hard to know which one the prediction is for.

[1] or the same static branch but with different missing history

To mitigate the accuracy loss, prior work generates multiple predictions, one for each possible path. Given N branches are skipped, there are 2^N possible missing history patterns since each branch can be either taken or not taken. When the prediction is finally needed, the missing history (available after all the intermediate branches are predicted) is used to select which of the 2^N predictions to use. While prior work successfully hides the prediction latency with minimum loss of accuracy, they incorrectly assume that all the missing history patterns are likely to be seen and thus generate 2^N predictions.

This design is impractical for ahead distances greater than one. Experimental results show that an ahead distance of 5 is necessary to cover the entire prediction latency (since, on average, more than one branch is fetched per cycle). A normal TAGE predictor reads out from the bimodal table, 6 short history tables with 12-bit entries, and 15 long history tables with 16-bit entries, resulting in 314 bits total for each prediction; however, its ahead version based on the design suggested by Seznec [55] would need to read out 10,048 (32x314) bits per prediction (for an ahead distance of 5). My analysis shows that covering the entire prediction latency incurs a 14.6x predictor energy overhead based on this design.

Furthermore, the problem worsens as the need to increase predictor capacity continues to grow to fit the current code footprint, which increases predictor latency and would require larger ahead distances. Because this design reads out exponentially more bits per prediction as ahead distance increases, per-prediction energy also increases exponentially. While it might be possible to use this design to hide 1 cycle of latency, hiding the full prediction latency is impractical as the branch predictor contributes to a significant proportion (3-4%) of the core power.

2.2.3 Decoupled Frontend

While the decoupled frontend[47] was originally proposed for instruction prefetching, it also hides the predictor latency if the predictor runs sufficiently ahead of fetch.

It uses a queue between the Branch Prediction and Fetch stages to buffer fetch addresses generated by the Branch Predictor. This is commonly called the Fetch Queue. If the predictor is running far enough ahead to buffer multiple fetch addresses in the Fetch Queue, a flush from the overriding predictor does not stall the rest of the frontend.

However, when there are not enough entries in the fetch queue, the decoupled frontend cannot hide the latency of the main predictor. My experiments show that 11.14% of the total number of early flushes are not completely hidden by the decoupled frontend and terminate a fetch packet early. The following three scenarios can cause this:

- After a backend redirect, the fetch queue is flushed and thus empty. During the next few cycles, any delay in the prediction pipeline is exposed to the fetch unit and adversely affects performance.
- When the program is in a region with a high misprediction rate from the single cycle predictor, the queue can become empty due to overriding predictor flushes. Even if the fetch queue is full the rest of the time, these regions with frequent flushes limit how fast instructions are delivered to the processor backend, hurting performance.
- When the program is in a region with a high taken branch density, the predictor cannot operate at peak bandwidth. This limits predictor bandwidth, making it hard to fill the fetch queue.

2.3 Number of Missing History Patterns

While prior work starts with the assumption that every missing history pattern needs to be considered, I start by asking the question: **How many missing history patterns are observed at program runtime for a specific ahead history and**

PC? If the number of patterns is small, I can leverage this fact to implement an efficient ahead predictor.

2.3.1 Experimental Results

I examine the benchmarks in the SPEC CPU2017 suite to answer the above question. For each branch on the correct path, I collect the control flow leading up to the branch and PC, along with the PC of the next 5 branches[2]. The control flow is captured via a global history register similar to the one used in the 2-level branch predictor[65]. The history and PC pair represent the ahead history and PC in my experiment. The sequence of the next 5 branches represents the missing history patterns observed. For each distinct control flow, I count the number of unique patterns for the next 5 branches. I use no history, 32 bits of history, and 64 bits of history, as shown in Fig. 2.2.

There are 32 (2^5) possible paths for the next 5 branches, assuming they are direct conditional branches[3]. However, significantly fewer patterns are observed in my experiment. I note a few important observations:

First, even when no history is used, the number of patterns observed is far less than the theoretical maximum. This is because some branches are inherently biased towards taken or not taken during a phase of execution or the entire program runtime.

Second, using history can drastically reduce the number of patterns observed. When no history is used, more than 4 patterns are observed 35.9% of the time on average. But when 64 bits of history are used, I observe more than 4 patterns only 1.48% of the time.

Third, benchmarks with high MPKI exhibit more patterns compared to benchmarks with low MPKI. Mcf, deepsjeng, leela, and xz come under this category. This

[2]I chose 5 branches for this experiment as this is the ahead distance used in my design

[3]If any of the 5 branches are unconditional, then the number of patterns can be infinite in theory

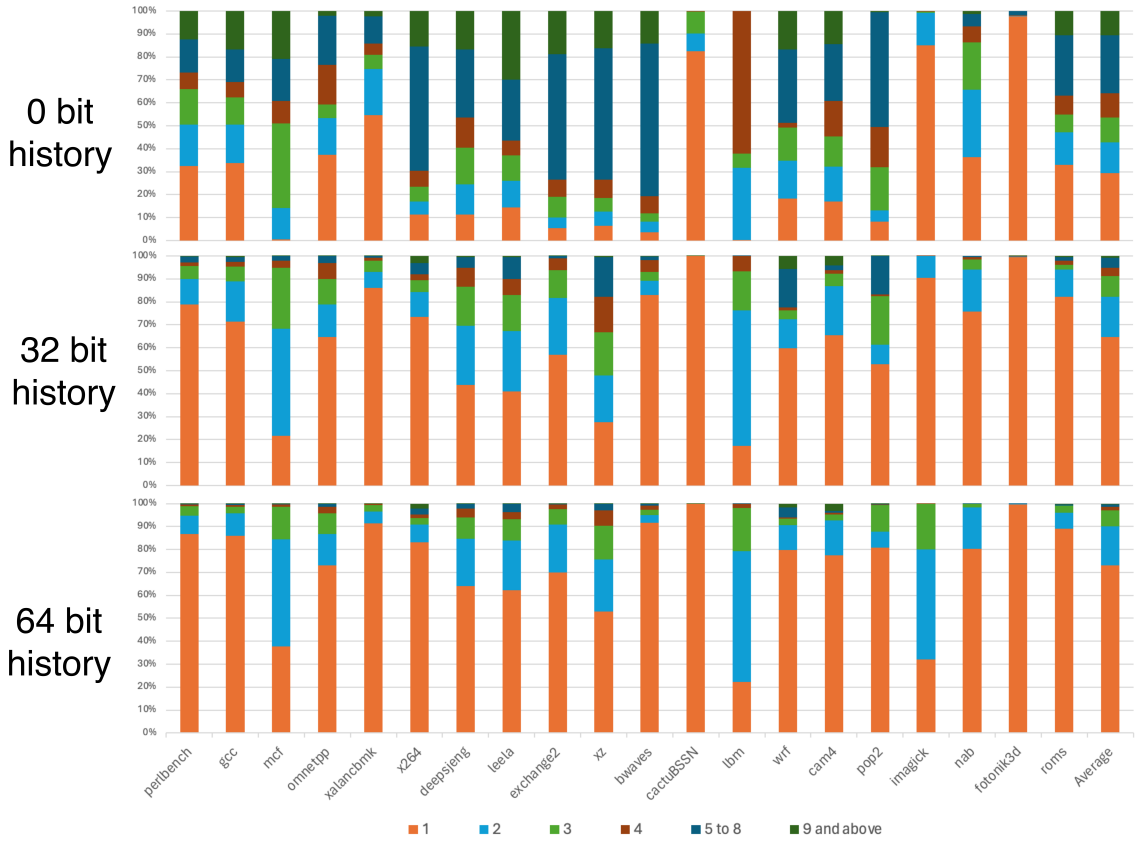

Figure 2.2: Number of Patterns Under Control Flow

is because the predictability of intermediate branches determines how many patterns are seen, as discussed in the following section.

2.3.2 Tying Everything Together

Fundamentally, most predictors work by assigning a saturating counter to each control flow leading up to a branch. At the time of prediction, the predictor looks for the counter assigned to the current control flow. As stated above, when the next N branches are all predictable, the current control flow only leads to one path, resulting in only one control flow N branches later. Thus, the prediction accuracy stays the same whether the ahead history or current history is used. This can be achieved by training the counter attached to the ahead history with the direction of the branch

to be predicted.

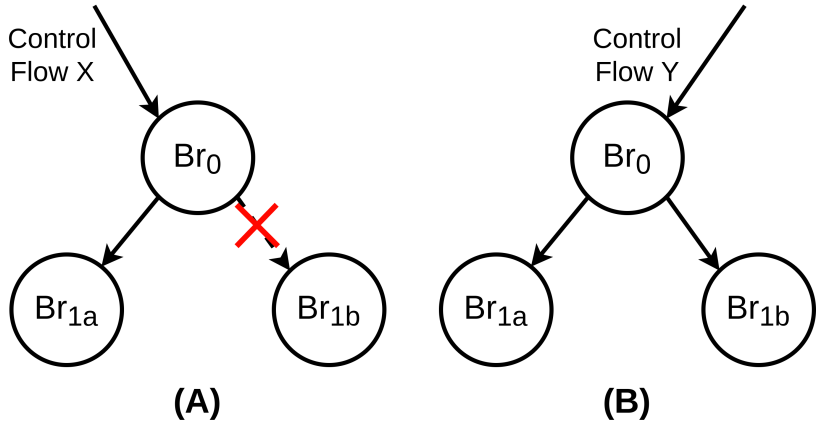

Figure 2.3: Control Flow Examples

However, if the branch skipped when using ahead history is unpredictable, prediction accuracy can drop significantly. Example B in Figure 2.3 illustrates this. Assume the predictor arrives at Br_0 under a different control flow Y that makes Br_0 unpredictable. In this case, then both Br_{1a} and Br_{1b} are possible targets of Br_0 under control flow Y, as shown in Fig. 2.3-B. If ahead prediction with an ahead distance of 1 is used here (using the PC and history at Br_0 to predict the next branch), the prediction counter associated with the control flow Y is trained by both Br_{1a} and Br_{1b}. If Br_{1a} and Br_{1b} go in the opposite direction, then both become unpredictable with the history at Br_0 even though Br_{1a} and Br_{1b} could be very easily predicted with current history (which contains the direction of Br_0).

Unlike other forms of aliasing in branch prediction that come from the history folding mechanism and the hashing function, this form of aliasing comes from inadequate information in the ahead history. In summary, predicting using ahead history works when only predictable branches are skipped. When unpredictable branches are skipped, the predictor cannot identify which path it is on, adding additional aliasing to the predictor. To remove the aliasing, the predictor must use the information missing from the ahead history, which I will refer to as missing history.

In the benchmarks I evaluated, most branches tend to be predictable. As seen in Fig. 2.2, 71% of control flows only lead to one path after skipping 5 branches as the skipped branches tend to be predictable. More than 4 patterns are seen when the missing history contains multiple unpredictable branches, but this is rare (1%). I use this insight to drive my ahead predictor design.

2.4 Designing an Efficient Ahead Predictor

Similar to prior work [23, 55], I generate multiple predictions from the ahead history and pick between them using the missing history. However, unlike prior work that accounts for all possible missing history patterns, I take advantage of the fact that only a few paths show up at runtime. I leverage two TAGE characteristics: 1) TAGE internally already reads out multiple counters, one from each history length, and 2) TAGE handles conflicts during allocation between different counters in the same table by promoting the allocation to a higher history. I use these two characteristics by distributing entries corresponding to different missing history patterns across different TAGE tables. I identify the missing history pattern that a counter belongs to with an additional tag field, the secondary tag. Fig. 2.4 shows the layout of a TAGE entry in my new design. Note that T0 (the bimodal table inside TAGE) remains untagged.

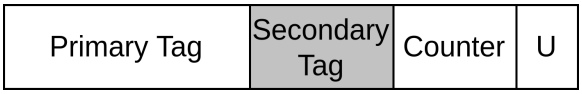

Figure 2.4: TAGE Entry with Secondary Tag

TAGE generates an index and a tag for each history length based on the PC and history. The selection logic picks the prediction from the matching table with the longest[4] history. In my design, the index and the primary tag are computed with the ahead history and ahead PC. A matching primary tag indicates the counter is intended for one of the previously observed missing history patterns under that ahead

[4]or second longest if alt-pred is used

history. A mismatch in the main tag means the entry corresponds to a different ahead history and should be ignored. My ahead predictor reads out one counter per table, similar to baseline TAGE. I duplicate the selection logic to identify the longest (or the second longest) matching counter **for each possible value of the secondary tag in parallel**. If no primary tag match is found, the prediction from T0 table is used. When the prediction is finally needed, the secondary tag is computed based on a hash of the missing history and is used to pick the final prediction. The secondary tag width determines the number of patterns my predictor can distinguish and is independent of the ahead distance.

Even though I duplicate the selection logic and generate multiple predictions, the number of bits read out from each table only increases by the width of the secondary tag. This number is far less than prior work [55] and does not increase exponentially with the ahead distance.

My ahead predictor design is shown in Fig. 2.5. It includes 26 history lengths (T0 through T25) similar to baseline TAGE. I use a 5-bit secondary tag as my design uses an ahead distance of 5 to cover a 3-cycle prediction latency. The next paragraph shows an example to help outline the prediction process.

2.4.1 Predicting a Branch: An Example

Given a particular ahead history, the predictor generates indices corresponding to the entries highlighted in red in Fig. 2.5. These entries are then read out and a tag comparison is performed. In the figure, there are 3 hits (based on the primary tag) in tables T1, T2, and T21. These entries correspond to different missing histories (secondary tags 1 and 31). The selection logic for each missing history value remains the same as baseline TAGE. For example, the selection logic for missing history 1 only sees a hit for the counter at T2 and generates a prediction based on its value. Similarly, the selection logic for missing history 31 sees hits in tables T1 and T24. The rest of the selection logic groups see no table hits and use the prediction supplied

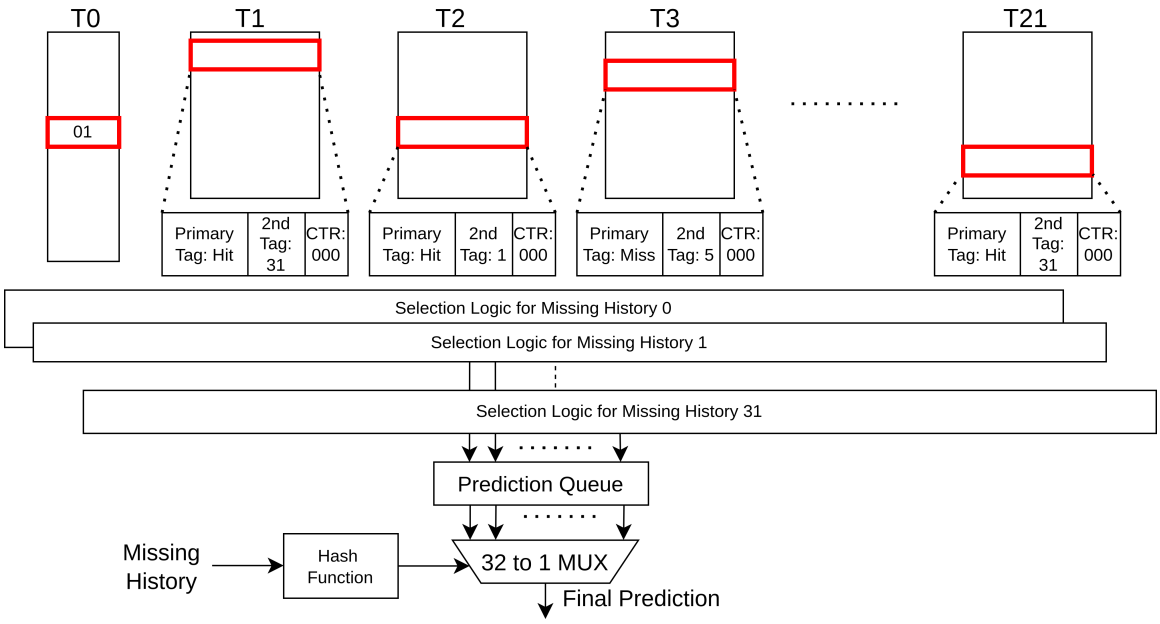

Figure 2.5: Ahead 2-Tag TAGE Prediction Example

by the bimodal table (T0). Note that even though the entry out of T3 has a missing history tag of 5, a mismatch in the primary tag indicates that the entry is not meant for the current control flow and should be ignored. Once the intermediate branches are resolved, I compute a hash based on the direction and target of the intermediate branches. The result of the hash would be the secondary tag for the branch I am predicting and determines the final prediction.

2.4.2 Selection Function/Secondary Tag

Prior work uses the directional history of the missing branches to pick between the generated predictions. For example, if the two missing branches were TAKEN, NOT TAKEN, then the secondary tag would be 10. This approach, however, has two major shortcomings. First, it cannot handle indirect branches as they could have multiple different targets but always a taken direction. Second, using the direction directly couples the length of the secondary tag with the ahead distance and makes it difficult to increase the ahead distance. I solve these problems by hashing together

the targets of the missing branches. The generated hash length is independent of the ahead distance and allows increasing the ahead distance without increasing tag width (see Section 2.6.4). In fact, **even using 1 bit of tag is enough to provide 2.2% performance and incurs only 20% of the corresponding area and energy overhead** compared to using a 5-bit secondary tag.

The selection function is computed based on the algorithm described in Fig. 2.6 and does not depend on any of the TAGE outputs for the current prediction. It is implemented as a regular 32-to-1 MUX (unlike the priority MUX logic at the end of TAGE). This results in lower latency and can be done in a single cycle. A software implementation of the hash function is described below.

```
for every branch skipped {
    addr = branch.pred_target
    selection = selection XOR addr[6:2] XOR addr[11:7]
    selection = ROTATE_RIGHT_BY_1(selection)
}
```

Figure 2.6: Missing History Hash Algorithm

2.4.3 Updating the Predictor

When a branch resolves, it follows the baseline TAGE update algorithm to update the counter and the usefulness bit based on the entries that provided the prediction. If an allocation is required, I again follow the baseline algorithm to find an existing entry to replace. The allocated entry is populated with the appropriate secondary tag to its corresponding value. Note that if a branch wants to allocate to an entry already containing a useful entry with the same primary tag but a different secondary tag (i.e., a different pattern corresponding to the same ahead history), the allocation is promoted to the next table. This follows the same algorithm that TAGE uses when dealing with conflicts.

2.4.4 The Importance of Having Few Patterns

Baseline TAGE relies on its allocation algorithm to place each counter in the table best suited for that branch (based on the history length required to predict that branch). During an allocation conflict, TAGE takes advantage of the fact that each table uses a different history length to compute the index. Since the histories used are different for each table, **entries that conflict in one table are unlikely to conflict in other tables**. This allows TAGE to minimize conflicts across the tables. In my design, indices are calculated using the ahead history and PC. This forces counters with the same ahead history but different missing history patterns to always use the same input to compute the indices for every table. As a result, they always conflict with each other in every table because they all have the same index for each table. The conflicts in allocations force these counters to reside in different tables. However, if there are too many patterns, these counters may experience many unnecessary promotions to a higher history, even if they could easily be predicted from a lower table using a shorter history. This increases the capacity pressure on the higher histories tables and hurts accuracy. However, because there are only fewer than 3 patterns most of the time (97%), these conflicts are minimized and do not significantly impact prediction accuracy.

I compare the prediction accuracy of branches in my ahead predictor with baseline TAGE. I categorize each branch based on the number of missing history patterns its corresponding ahead history observes, and the results are shown in Table 2.1. This shows that branches with only a few missing history patterns (1-3) see very little degradation in accuracy with my ahead predictor. For branches with more patterns (> 3), the accuracy drop is more significant due to the additional conflicts as explained above. However, since this is rare, its impact on overall accuracy is small: for all branches, my ahead predictor only decreases the prediction accuracy by 0.067% compared to baseline TAGE across all benchmarks in SPEC2017.

Number of Missing History Patterns	1-3	4-6	7 and above	Overall (All Branches)
Misprediction Rate Delta	0.065%	0.15%	0.16%	0.067%

Table 2.1: Accuracy Diff between Ahead Predictor and Baseline

2.5 Integration with Core

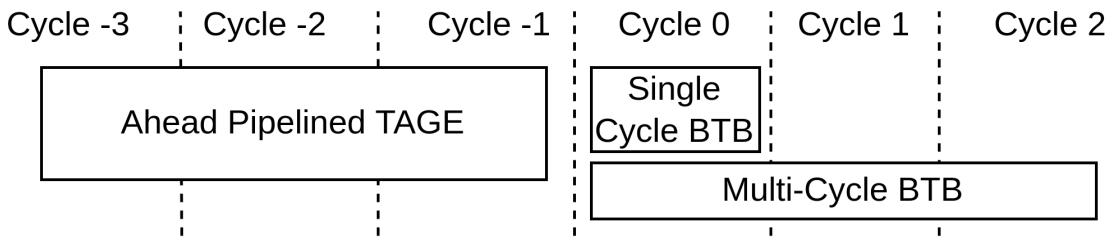

Figure 2.7: Prediction Timing Diagram

Fig. 2.7 shows the timing diagram for predicting Branch N. Cycle 0 is when the PC and full history of branch N become available and when the prediction of Branch N is needed. The ahead predictor starts predicting branch N several cycles earlier, using the PC and history of Branch 0, where N is the ahead distance. This prediction (in most cases) becomes available in cycle 0. In cycle 0, the PC of branch N is used to access both the single-cycle BTB and the multi-cycle BTB for the target of Branch N. The target from the single-cycle BTB and the ahead prediction results are used to determine the next fetch address. On a single-cycle BTB miss, the branch is assumed not-taken until the multi-cycle BTB access is completed, and a late flush is issued if the prediction was taken. Note that the single-cycle BTB and multi-cycle BTB use the current PC (PC of branch N), thus are not ahead pipelined and operate identically to the baseline.

Fig. 2.8 shows the ahead prediction pipeline. The ahead predictor uses the current PC and current history to generate a prediction for the branch exactly N

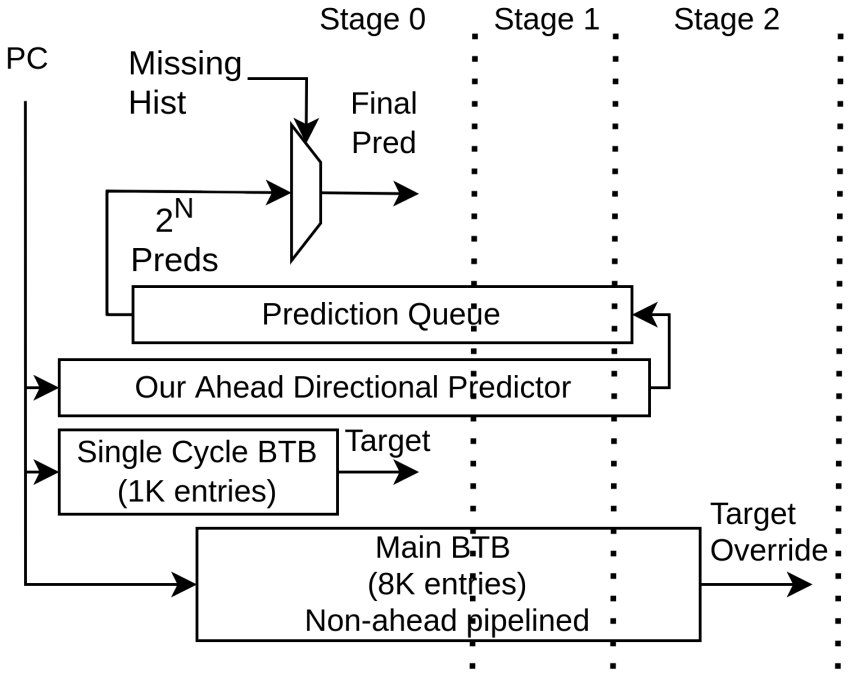

Figure 2.8: Prediction Pipeline

ahead (branch N is predicted with the PC and control flow at branch 0). The ahead predictor generates 2^M predictions (for an M-bit secondary tag), which are saved to the prediction queue after 2 cycles.

The prediction queue entry is read out when the PC reaches branch N. At this point, intermediate branches have been predicted, and their directions are used to pick the final prediction for branch N among the 2^M predictions.

2.5.1 Single-Cycle Override

Both prior work [55] and my design suffer from counter duplication for branches that could have been accurately predicted with a short history (less than the ahead distance). For example, in a baseline non-ahead pipelined TAGE, if a branch is biased to be either taken or not taken, a single entry in table T0 can accurately predict the branch. However, when the predictor is ahead-pipelined, a counter is needed for each

of the possible ahead histories that lead to this branch. If there are multiple control flows that lead to this branch, multiple counters are needed, making the branch much harder to predict. I mitigate this problem by keeping the baseline single-cycle predictor: the 2-bit counter in each entry of the single cycle BTB. This predictor helps deal with these branches as it only uses the current branch PC to generate its prediction. I allow predictions from the single cycle predictor to override the ahead predictor if they are more confident.

A 3-bit counter per branch in the single-cycle predictor is used to track the usefulness of the predictor entry. The counter is incremented when the bi-modal prediction is correct and the ahead prediction is wrong. The counter is decremented when the ahead predictor is correct and the bi-modal predictor is wrong. The counter stays the same if they are both correct or both incorrect. The counters are updated when the branch retires. If the counter value is over a threshold of 2, the ahead prediction is ignored and single-cycle prediction is used. Overall, this provides a 1% performance benefit across all of SPEC2017 benchmarks.

2.5.2 Prediction Queue Management

The prediction queue buffers all the predictions generated by the ahead predictor. It is implemented as a circular buffer. Each entry in the prediction queue has one ready bit and one bit for each prediction generated (33 bits total for a secondary tag width of 5, 1 bit for the valid bit, 1 bit for each of the possible 2^5 secondary tag values). An entry is allocated to this queue with ready bit set to zero. The ahead predictor populates the predictions in that entry when generated, and sets the ready bit to one. The prediction queue is controlled with three pointers: an allocation pointer, a read pointer, and a write pointer. Predictions are read out of the entry pointed to by the read pointer when the branch that needs these predictions is seen. This also increments the read pointer by 1. At the same time, the prediction for a future branch is started and a new entry is allocated in the prediction queue. This increments the allocation pointer by 1. When the predictions from TAGE are ready,

they are recorded in the entry pointed to by the write pointer. This action also sets the ready bit, and increments the write pointer.

Note that when the machine starts, the first N branches do not have predictions from the ahead predictor where N is the number of branches skipped. To account for this, the read pointer is initialized to 0, the write and allocation pointers are initialized to N-1. The size of the prediction queue is the sum of the maximum number of in-flight branches and ahead distance. This guarantees that the prediction queue never overflows and does not introduce any additional stalls.

2.5.3 Late Predictions

Modern processors predict up to the first taken branch per cycle, thus the exact number of branches that are seen in 3 cycles is not fixed. Although an ahead distance of 5 branches covers 3 cycle latency most of the time, it is possible for a prediction to arrive late. In the event that a prediction arrives later than it is needed, the result from the single-cycle predictor is used. When the ahead prediction becomes available, I compare it against the single-cycle prediction. If this prediction agrees with the bimodal prediction, then no further action is needed. Otherwise, the predictor can issue an early pipeline flush, and the prediction pipeline can start from the new address on the following cycle. Alternatively, a simpler but less performant design can be done by always stalling the prediction pipeline when the ahead predictor is late. This still provides 2.4% of IPC improvement over baseline.

2.5.4 Prediction Pipeline Restart

My design handles misprediction flushes by manipulating the read, write, and allocation pointers to the prediction queue. For each branch, I checkpoint the read and allocation pointers at the time of prediction. If this branch is mispredicted, the allocation pointer and read pointer are moved to their respective checkpointed values plus 1. The new write pointer is set to be the same as the allocation pointer since

all prior predictions have finished by then. By doing so, all entries in the prediction queue where the predictions are made with ahead history containing the mispredicted branch are effectively removed (as the mispredicted branch only shows up in the ahead history 5 branches later). Notice that after the flush, there are 4 predictions after the mispredicting branch remaining in the queue. These predictions were made with an ahead history that did not contain the mispredicted branch.

Similar to prior work [55], my predictor does not incur any extra misprediction penalty and can start immediately after a misprediction flush. The main idea here is that the predictions for the branches on the new paths are already computed via the ahead history before the misprediction as they do not use the direction of the mispredicted branch. By simply buffering up the predictions (1 bit per missing history pattern value per branch) until the branch they were intended for retires, I achieve high prediction accuracy right after a pipeline flush at minimum overhead.

Fig. 2.9 shows what happens if Br_X is mispredicted by my ahead predictor. I show the state of the prediction queue when Br_X enters the prediction pipeline, when the misprediction is detected, and after the recovery is finished. Notice that after the recovery, the prediction for Br_{X+1} still remains in the prediction queue and can be read out immediately. Note that while the relative positions of the read pointer and the alloc pointer are always separated by the ahead distance, the write pointer can be anywhere between the read pointer and the alloc pointer depending on the number of branches encountered per cycle.

2.5.5 Critical Path Analysis

In the cycle when a prediction is needed, an entry is read out from the prediction queue and the selection logic picks the final prediction. Unlike the TAGE internal selection logic, my final selection logic does not depend on the predictions themselves, thus it can be done in parallel with reading from the prediction queue. The critical path of delivering the prediction includes the read from the prediction

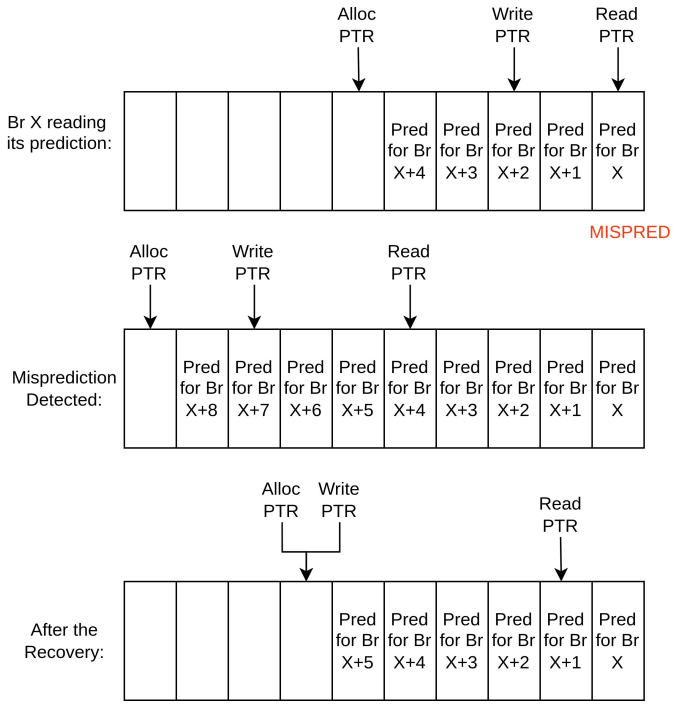

Figure 2.9: Prediction Queue Recovery Example

queue and the propagation delay through the muxes, and should not increase the critical path of that stage.

2.5.6 Hardware Overhead Analysis

The main hardware overhead is the secondary tag in the TAGE predictor. A 5-bit secondary tag introduces an additional 18.75KB of storage in the predictor. The prediction queue is sized to be 133 entries. Each entry in the prediction queue is 33 bits, resulting in 549B of storage overhead. The additional counter in the single-cycle predictor is 3 bits per entry and results in 384B storage overhead. Overall, my design adds 19.65KB of area overhead. Note that I provide the comparison between my design and a large baseline TAGE to match the storage in Sec. 2.6.7.

2.6 Evaluation

2.6.1 Methodology

Core	3.2GHz, 16-wide issue 512 Entry ROB, 256 Entry Reservation Station
Caches	32KB 8-way L1 I-cache & D-cache 4-cycle access 1MB 16-way LLC cache 18-cycle access, 64B lines
Memory	DDR4_2400R: 1 rank, 2 channels 4 bank groups and 4 banks per channel tRP-tCL-tRCD: 16-16-16
Single Cycle Predictor	1K-entry 4-way target buffer 2-bit saturating counter per entry
Main Predictor	8K-entry 4-way target buffer TAGE: 8K entry T0, 6 short histories, 15 long histories 10 1k-entry short tables, 20 1k-entry long tables Total Capcaity: 56.63KB up to 1 taken branch per cycle, 3-cycle latency
Predictor Bandwidth	Up to the first taken or 16 instructions
Fetch Bandwidth	Up to 16 instructions
Fetch Queue	8 Prediction Packets

Table 2.2: Simulation Parameters

To evaluate how my predictor design affects prediction accuracy and overall processor performance, I simulate the micro-architecture of an aggressive out-of-order core in an execution-driven cycle-accurate x86-64 simulator [3]. The system details for the baseline OoO core and additional structures for my ahead predictor are listed in Table 2.2. My baseline models a very aggressive OoO core that contains a multi-level predictor and a decoupled frontend.

I use TAGE as the main predictor as it is the most common predictor found in products today and is the main component of the TAGE-SC-L predictor [56]. The baseline TAGE predictor is configured exactly as the TAGE predictor from TAGE-SC-L in CBP5[1]. Ahead pipelining the statistical corrector (SC) is expensive because it requires multi-porting the internal tables. The loop (L) predictor is a small table

and can likely be looked up in a single cycle. Overall, SC and L only provide modest performance improvements (1.11%) over the baseline TAGE.

I use all applications from SPEC CPU2017 (speed) in my evaluation. I use the SimPoints Methodology [61] to generate up to 5 simpoints for each input set, with 200 million instructions per simpoint.

2.6.2 Results

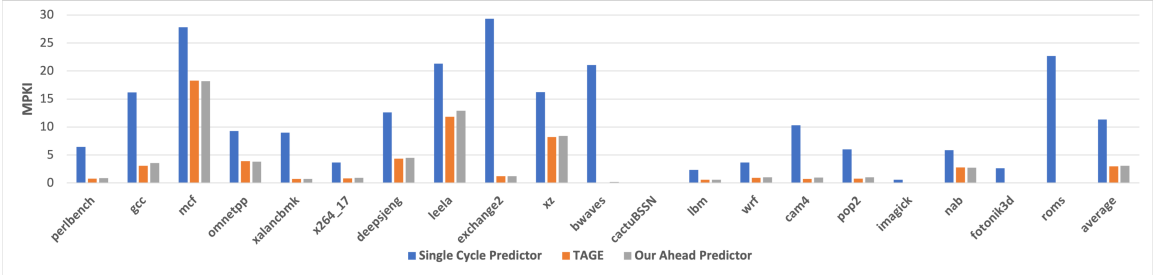

Figure 2.10: MPKI of Baseline TAGE and My Ahead Predictor

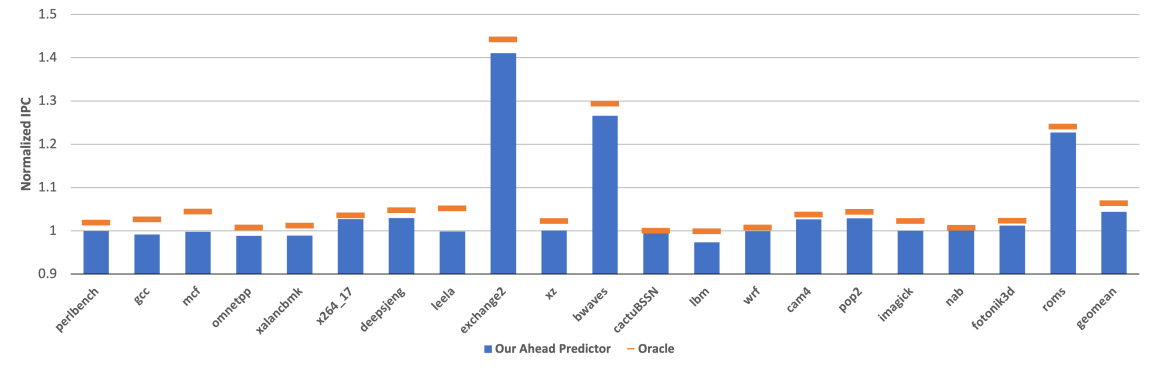

Figure 2.11: Normalized Performance Improvement

I first evaluate the performance impact of my ahead predictor compared to the baseline TAGE. Fig. 2.10 shows the MPKI of baseline single cycle predictor, baseline TAGE and my ahead predictor. On average, my predictor is within 0.1 MPKI of the baseline TAGE. Fig. 2.11 shows the normalized performance of using my ahead predictor compared to a baseline out-of-order core with multi-level prediction. I also

compare my scheme against an oracle where the TAGE latency is reduced to 1 cycle and is used as the single cycle predictor. Overall, my design provides 4.4% geomean IPC improvement. This is within 68% of an ideal solution (single cycle TAGE that provides 6.42% performance improvement) while still being physically realizable.

Exchange, bwaves, and cams benefit the most because they are bound by the instruction supply and do not suffer from much backend pressure. Moreover, these benchmarks suffer from high single cycle predictor MPKI, but have low MPKI on TAGE, which makes ahead prediction a good choice for them because they do not exhibit many missing history patterns. Leela, mcf, and xz show some performance improvement with an oracle single-cycle TAGE, but my design is unable to capture all the missing history patterns as there are many clustered unpredictable branches in these benchmarks. In omnetpp and xalancbmk, my ahead predictor has a better MPKI but shows worse performance. This is because wrong path instructions in these benchmarks help prefetch data that is eventually useful, and removing these mispredicting reduces this prefetching effect. Gcc loses performance because of the large number of static branches. This increases the capacity pressure on my ahead predictor as explained in Sec. 2.5.1, decreases the predictor accuracy, and adversely affects performance.

Using TAGE-SC-L as the baseline: Compared to a non-ahead pipelined TAGE-SC-L, my TAGE-only ahead predictor implementation provides 3.3% IPC improvement.

2.6.3 Energy Comparison against Prior Work

The main difference between my predictor design and prior approaches is that my design generates multiple predictions under the same ahead history more efficiently. Prior work generates predictions by reading consecutive entries out of each prediction table, leading to a drastic increase in the number of bits read out per prediction. This increases exponentially with the ahead distance, making it infeasible

to implement for large ahead distances. My design reads out one entry per table, but each entry contains a few more bits. These bits correspond to the secondary tag which scales linearly with ahead distance.

Both my design and prior work require duplicating the selection logic with each possible missing history value. Since the selection only involves comparators, MUXes, and reading from the small alt-pred table (16 entries), the energy required from this is significantly less than the table reads. Thus I approximate the energy required per prediction by measuring the energy of the table reads.

TAGE table reads account for the majority of the energy consumed per prediction. The number of bits read out is directly correlated with the energy needed to access the prediction tables. Thus, the energy consumption of my design increases linearly with the ahead distance while the energy of prior work[55] increases exponentially. I use Cacti[40] to simulate the energy required for each prediction. The baseline model consists of a bimodal table (8K entries with a 2-bit port), 6 short history tables (1K entries with a 12-bit port), and 15 long history tables (1K entries with a 16-bit port). As ahead distance increases, prior work would double port sizes in each table. In my ahead predictor, as ahead distance increases, the port size only increases by 1 in each table (except for the bimodal table). Fig. 2.12 shows the per-prediction energy (normalized to baseline TAGE) required for different ahead distances. The numbers show that the scaling is much better for my design which makes it practical to implement even for large ahead distances.

2.6.4 Secondary Tag Size

A longer secondary tag increases the area and energy overhead; however, it makes it easier to differentiate between missing history patterns for the same ahead history. This reduces the amount of aliasing caused by ahead prediction. A shorter secondary tag is more efficient and adds less area overhead. I evaluate my design at an ahead distance of 5 with 0 to 9 bits of secondary tag. A tag width of 0 means

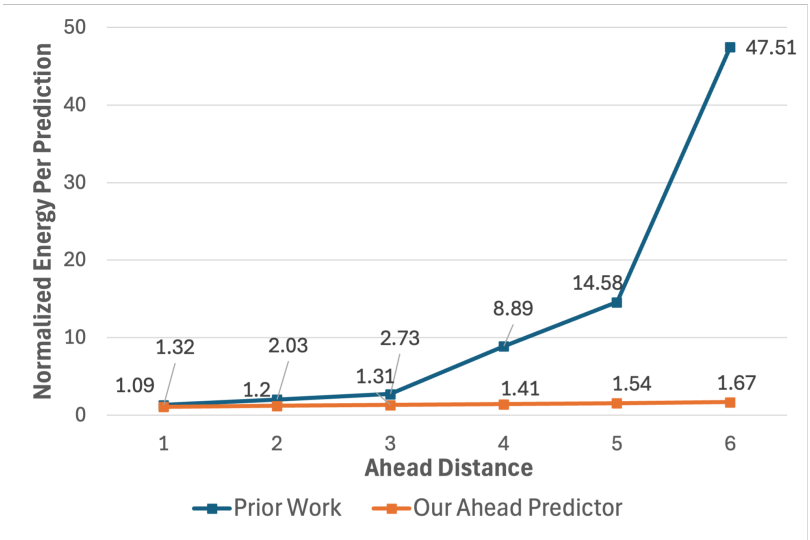

Figure 2.12: Normalized Energy vs. Ahead Distance

that the predictor just uses ahead information and does not use any missing history information.

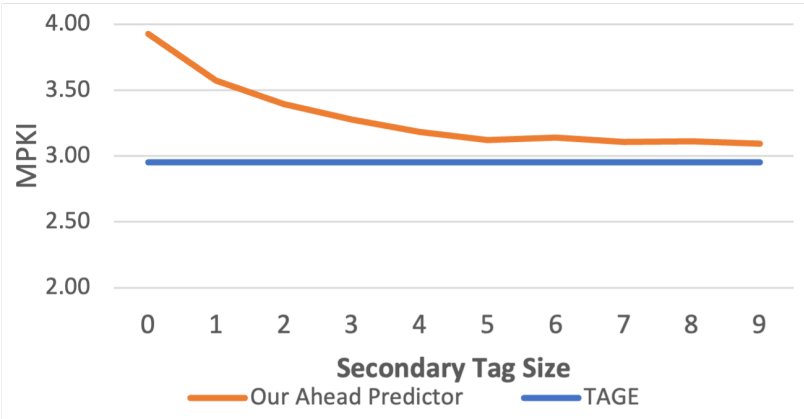

Figure 2.13: MPKI vs Secondary Tag Size

Fig. 2.13 shows the MPKI of my predictor for different secondary tag widths. A tag width of 0 suffers the most aliasing and has a much higher MPKI than baseline TAGE. As the tag width increases, the amount of aliasing reduces and so does the MPKI. The benefit of adding more tag bits shows diminishing returns: After a tag width of 4, the MPKI reduction is minimal. In terms of performance, using 1 bit

of tag is enough to provide around half the benefit (2.2% performance) which shows performance is even more skewed when it comes to the diminishing returns. I chose to go with a 5-bit tag to get the highest possible performance, but the tag width being decoupled from the ahead distance provides a wide design space to optimize the ahead predictor design.

2.6.5 Ahead Distance

Fig. 2.14 shows the MPKI of my ahead predictor and prior work as ahead distance increases from 3 to 7. With a longer ahead distance, more missing history patterns are exposed. My solution performs slightly worse than the prior work, as prior work considers all possible missing history patterns. However, the per-prediction energy increases exponentially as explained in Sec. 2.6.3, making prior work impossible to be implemented.

My baseline uses a 3-cycle TAGE predictor. Experiments show that an ahead distance of 5 can cover the entire prediction latency 91.3% of time. Fig. 2.15 shows the normalized IPC of my predictor as the ahead distance increases (the secondary tag increases with the ahead distance). In my design, an ahead distance of 6 covers almost all of the predictor latency.

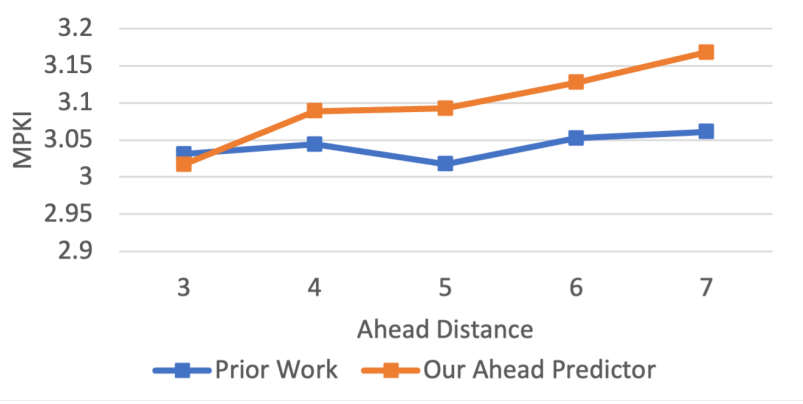

Figure 2.14: MPKI vs Ahead Distance

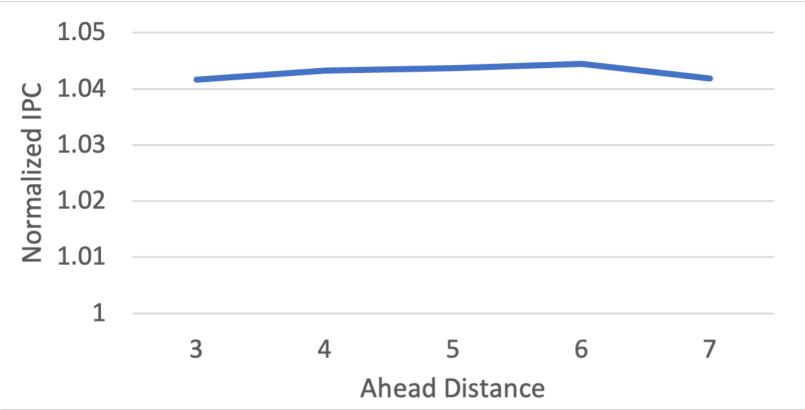

Figure 2.15: Normalized IPC vs Ahead Distance

2.6.6 MPKI vs Number of Tables Read

Fig. 2.16 shows the MPKI of my ahead predictor as the number of tables read per prediction decreases from 21 to 14. I use the banking interleaving feature introduced in the latest version of TAGE, where the number of histories is less than the number of physical tables. This allows us to change the number of tables read per prediction without changing the underlying capacity of TAGE. In this experiment, I gradually remove the number of 2-way histories in TAGE until all histories are 1-way. More tables read per prediction help with the distribution of counters from the same ahead history. However, this effect does not show up significantly until only 14 tables are read per prediction.

2.6.7 ISO-Area Comparison

The secondary tags incur 18.75KB of extra storage. If the same storage is applied to the baseline TAGE at the same latency, it can only achieve 0.13 MPKI and 0.19% performance improvement over baseline, much lower than what my ahead prediction scheme offers.

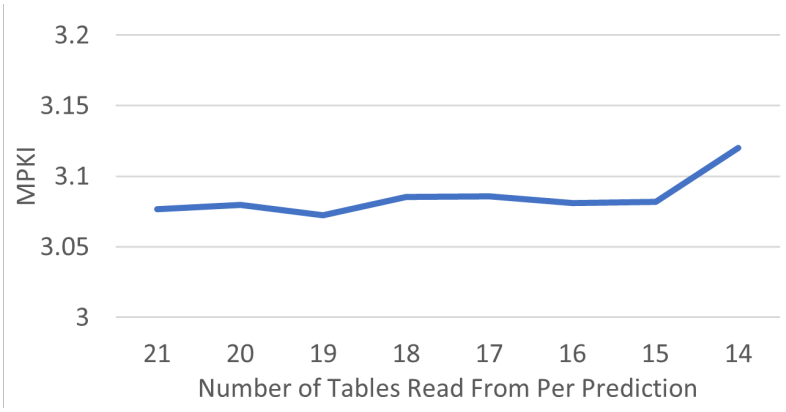

Figure 2.16: MPKI vs Number of Histories

2.7 Related Work

Unlike the TAGE predictor that attaches a prediction counter to a particular control flow, Perceptron Branch Predictor [21] uses a weight for each bit in the history representing its contribution to the overall prediction and sums all the weights up at the time of prediction. The latency of Perceptron is later improved [19, 24] by computing the sum as the directions of the previous branches are made available. Therefore, only one addition is needed during the last cycle. However, this approach only applies to perceptron.

Loop cache allows the processor to replay instructions in a short loop, usually at full fetch/issue width. In this case, the predictor is bypassed completely, and no predictor latency is observed. There have been many adoptions from industry, but these techniques take a long time to lock on to the loop, and can only be applied in very limited cases.

Recent work [11] has shown that predictor storage in current designs does not fit the application footprint of server workloads. While there have been multiple proposals to mitigate the capacity pressure on the BTB [27, 28, 32, 46, 62], very few works have been published on predictor capacity. Whisper [29] uses offline profile information to determine a static prediction function for predictable branches. In doing

so, predicting these branches does not require capacity in the main predictor, leaving more capacity for other branches. Last-Level Branch Predictor[49, 50] proposes using secondary storage to back up the main predictor, and a prefetcher to manage the secondary storage. While my design can eliminate the prediction latency, it cannot increase the prediction size indefinitely for 2 reasons. The larger predictor size leads to a longer ahead distance and can negatively impact performance as explained in Section 2.6.5. In addition, the physical location of the main predictor is near the processor frontend, where the on-chip area is extremely contested. A prefetcher approach like the Last-Level Branch Predictor [49] can enable larger predictor designs beyond what an ahead prediction approach can provide. I believe all three approaches can be combined in the future.

My ahead prediction scheme increases the effective predictor throughput. This is because each early pipeline flush from the disagreement between TAGE and the single-cycle predictor effectively stalls the prediction pipeline for 2 cycles, making the prediction throughput only 1/3 of its peak throughput. FDIP [47] relies on the addresses generated by the branch predictor to prefetch into the I-cache. A faster prediction unit allows the predictor to run further ahead, providing more opportunity for prefetching. APF [12], CDF [10], Precise Runahead Execution [41], and TEA [13] use the main predictor to generate the critical/runahead control flow. A faster predictor would enable all of these works to run further ahead and extract more performance.

2.8 Conclusion

Modern branch predictors are large and complex. They cannot predict branches within a single cycle, introducing bubbles in the pipeline and hurting processor performance. Ahead prediction is a widely proposed solution to this problem but drastically increases prediction energy as exponentially more entries are read out for each branch skipped, making building such a predictor impractical.

I show that only a few missing history patterns are observed in the program's

runtime. Using this insight, I present a new approach for building ahead predictors that does not require reading exponentially more entries for large ahead distances. My ahead predictor provides a 4.4% performance improvement while increasing power by only 1.5x, as opposed to prior designs that incur a 14.6x energy overhead. By hiding the predictor latency from the rest of the pipeline, my work allows for larger and more complex predictors and better pipelining width scaling.

Chapter 3: Achieving a Longer Effective History via Pruning Predictable Branches

3.1 The Problem

The accuracy of modern branch predictors often relies on capturing correlations that span hundreds or even thousands of dynamic branches. For example, TAGE-SC-L [56], the winner of CBP5, uses a 3000-bit global history register, while RUNLTS [31], the winner of CBP6, uses a 4316-bit history register. These long histories allow predictors to learn correlations between branches separated by thousands of instructions, capturing patterns that are crucial for minimizing mispredictions in complex workloads. Figure 3.1 shows that increasing the size of history register reduces MPKI, confirming that longer histories are desirable.

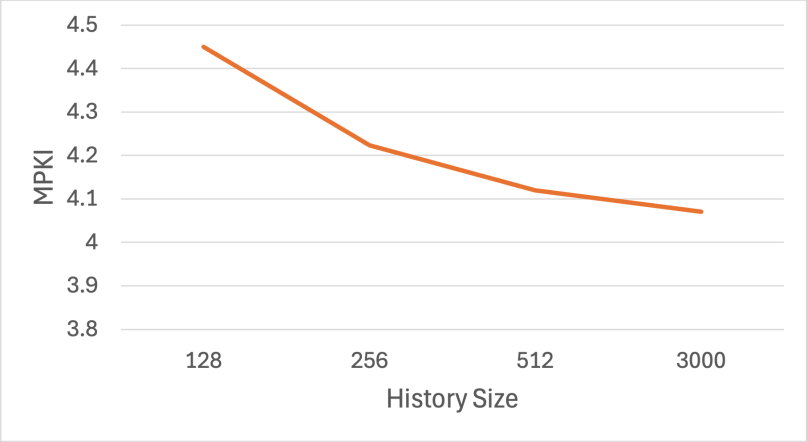

Figure 3.1: MPKI vs. History Length

However, as discussed in Section 3.2.3, simply increasing the size of the global history register comes at a high cost in storage, recovery latency, and energy, making straightforward history scaling impractical.

The key insight from my study of predictable branches offers a solution: predictable branches do not introduce new control-flow information and therefore do not

provide new information to the global history register. Consider Listing 3.1, a code example with 4 branches. Both branch 3 and branch 4 are predictable. A naive argument would suggest that B4, which strongly correlates with B3, requires B3 in the history to be predicted accurately. However, B3 is predictable precisely because its outcome is determined by B1 and B2, which are already in history. Any downstream branch correlated with B3 is therefore equally correlated with the existing history — making B3's history bit strictly redundant. By selectively excluding predictable branches from history updates, the predictor can effectively extend the history available for capturing meaningful correlations without physically building a longer history register, retaining high accuracy while reducing the overheads that traditionally make long histories impractical.

```
if (A) { ... }          // B1: hard, genuinely uncertain
if (B) { ... }          // B2: hard, genuinely uncertain
if (A && B) { ... }     // B3: predictable,
                        //     outcome deterministic based on B1 and B2
if (!A || !B) { ... }   // B4: correlated with B3
```

Listing 3.1: Predictable branch redundancy example

3.2 Background and Prior Work

3.2.1 TAGE

TAGE [59] is the most common branch predictor in modern processors today. Every predictor submission [6, 7, 15, 25, 31, 34, 39, 48, 58] in the 6th Championship Branch Prediction Competition [5] builds on TAGE as its core component.

On a high level, TAGE keeps multiple tables, each associated with a different history length. A new branch would be predicted with no history(out of the bimodal table), and then progressively predicted with higher history table after each misprediction until the branch stops mispredicting. At the time of prediction, TAGE uses the branch PC and the specified history length for each table to generate a tag and index for each table. Once that is done, all tables are read in parallel at the computed

index, and the tag stored in the entry is compared against the tag that was computed. A tag match (hit) indicates this entry is likely intended for the current branch under the current control flow. If there are multiple hits, TAGE picks the prediction from the longest history table that was a match.

In addition to the GHR, TAGE maintains a path history register (PHIST) that tracks the PCs of recently executed branches. While the GHR is long — spanning hundreds of bits — and captures deep control-flow context, the path history is short, typically tens of bits, and captures only recent control-flow information. Both histories are used to generate the index and tag during prediction.

3.2.2 Baseline History Update

While academic predictors typically assume one history update per branch regardless of direction, modern branch predictors update the history registers once per prediction packet. A prediction packet is a group of branches predicted together in a single cycle, terminating at the first taken branch encountered during prediction. Both path history and the global history are updated based on the hash of the branch PC and target PC. Updating once per packet on the taken branch provides equivalent information to updating on every branch: since a taken branch naturally delineates the boundary between packets, its occurrence implicitly encodes where all preceding not-taken branches fell, preserving the full control-flow information of the packet in a single update.

3.2.3 Why Naive History Scaling Is Difficult in Practice

Despite significant effort, no existing recovery mechanism scales gracefully to the history lengths that modern high-accuracy predictors require [31, 56]. Three methods are commonly used, each managing the tradeoff differently, but all sharing a fundamental limitation: their costs grow with history length in ways that make very long histories impractical.

3.2.3.1 Checkpointing

The checkpointing approach saves the entire state of the history registers at the time of prediction to a dedicated memory. On a misprediction, the processor reads the checkpointed history so that the resulting history reflects the current control flow. While simple in concept, this approach cannot support very long history sizes due to two reasons.

- Storage overhead: Each speculative branch requires its own history checkpoint. As processors today contain a large instruction window, the number of checkpoints required can be substantial. For example, to support a processor with an instruction window of 1000, the checkpoint has to be able to support up to 250 branches[1]. Supporting long history size such as 3000 bits as proposed in [56] would incur a storage overhead of 732KB worth of storage.

- Latency to read the checkpoint from the storage: When a misprediction is detected, the processor issues a pipeline flush so that the correct-path instructions can be fetched and sent to the processor backend. A larger checkpoint storage would increase the time it takes to read out the checkpoint, thus delaying the pipeline restart, adversely affecting the overall performance. Prior work[12] has shown that each additional cycle in the pipeline restart after a flush decreases the IPC by 1%.

3.2.3.2 Backwards Shifting

An alternative is to implement the history register as a shift register. The shift register is made of two parts, a visible region that is available to the branch predictor at prediction time, and shadow region that is used for recovery only. Each update would shift the history register forward, causing some bits to leave the visible region and move into the shadow region. On a branch misprediction flush, the bits

[1]Assuming 1 in every 4 instructions is a branch.

in the shadow region are shifted back as if the branch misprediction never happened. While this eliminates the need for a large checkpoint storage, it still suffers from high energy and long latency to recover to the correct history:

- Energy: Every history bit must be shifted on each update, leading to high dynamic power consumption, especially for long history registers.
- Latency to recover on a pipeline flush: The recovery process now involves a large rotator, where the rotation amount equal the (maximum number of in flight branches * the number of bits shifted per branch). This can add significant delays to pipeline restart, decreasing performance.

3.2.3.3 Circular Buffer

Another recovery mechanism stores history in a circular buffer with a head pointer. While this approach minimizes both checkpoint storage and recovery latency, it introduces significant latency at prediction time. Similar to the shift register implementation, the circular buffer is divided into two regions, a visible region and a shadow region. Each branch checkpoints only the head pointer during prediction. When a misprediction occurs, recovery simply moves the head pointer back to the checkpointed location, requiring no shifting or rotating during a pipeline flush. However, since the head pointer can point to anywhere in the circular buffer at prediction time, a rotator must support shifting by up to the full size of the circular buffer, often in the range of thousands of bits. This design drastically increases the prediction latency, making this implementation impractical.

Together, these issues illustrate why naively scaling history length — simply allocating more bits to the history register — is not a practical solution. Each recovery mechanism introduces its own tradeoffs between storage, recovery latency, prediction latency, and energy, and these costs grow with history length. This motivates mechanisms that allow the effective history length to grow without proportionally increasing the physical history register.

3.2.4 Prior Work

Prior work has recognized that not all branches contribute information to the global history. The Bias-Free Branch Predictor [16] observes that highly biased branches — those that are virtually always taken or not taken — are redundant in the global history. Since a nearly constant variable has near-zero variance, it cannot meaningfully correlate with any other branch, and therefore contributes no predictive value to the history register. By filtering such biased branches from the global history, the Bias-Free branch predictor increases the effective history length for perceptron-based predictors.

My work addresses a strictly broader class of redundancy. A biased branch is always predictable — its near-constant outcome is trivially determined regardless of history context. However, a branch need not be biased to be predictable. A branch whose marginal distribution is close to 50/50 can still be fully predictable if its outcome is already determined by the existing history. My predictability criterion therefore strictly subsumes the Bias-Free criterion — every branch that Bias-Free filters from updating the history is also filtered by my approach, but my approach additionally identifies branches that are contextually predictable yet not statistically biased.

3.3 Branch History Pruning Algorithm

Determining which branches to skip is more nuanced than simply filtering branches that are predictable. Several categories of branches require special consideration to ensure that skipping does not inadvertently degrade prediction accuracy. This section examines each case and discusses how to handle them, both analytically and empirically.

3.3.1 What to Skip?

Since the history update happens at the packet granularity, the prediction packet is eligible to skip its history update only if every branch within the packet — including the taken branch that terminates it — is eligible for skipping. If any branch in the packet is ineligible, the packet updates the history as normal. This ensures that no branch's control-flow information is lost.

3.3.1.1 Backward Branches

Consider the example in Listing 3.2, a simple loop that iterates ten times before exiting. Under normal history updates, the loop branch would modify the history at the end of each iteration, allowing the predictor to implicitly encode the iteration count and distinguish the final exit from earlier iterations. If the loop branch skips history updates, however, all iterations appear identical to the predictor — the program counter and global history are the same on every iteration, making it impossible to distinguish early iterations from the final exit.

This problem extends beyond loop branches to any backward branch that skips a history update, as a backward control transfer may cause execution to revisit the same static branches without any history change. Forward branches do not cause this problem — since branches after a forward branch would never have the same PC. Backward branches, however, can make distinct dynamic instances of the same static branch indistinguishable to the predictor, significantly degrading prediction accuracy.

```
loop_start: mov     ecx, 10         ; loop count = 10

            ; loop body

            dec     ecx             ; decrement counter
            jnz     loop_start      ; branch if not zero
```

Listing 3.2: Ten-iteration loop Example

To solve this problem, a global counter is introduced to keep track of the

number of backward branches since the last update to the history register. This counter is incremented by 1 on each backward branch that skips the history update and is cleared to 0 when any branch updates the history register. This counter allows the predictor to differentiate different dynamic instances of the static branch without having the backward branches update the history register. For each branch instance where the counter is non-zero, I use a virtual PC (VPC), a technique introduced by [30], to predict such a branch. The VPC of the branch is computed by adding the original PC to the global backward counter.

However, incrementing the PC risks the VPC colliding with the PC of another static branch, causing the predictor to alias two distinct static branches. To mitigate this, a history update is forced when the counter reaches a predetermined threshold, bounding the maximum VPC displacement. Table 3.1 compares three history update strategies for the loop in Listing 3.2. In the baseline (no skipping), each iteration produces a unique history H_1 through H_{10}, allowing the predictor to clearly distinguish the exit condition at iteration 10. Under naive skipping, all iterations share the same PC and the same history H. This means that the first 9 iterations become indistinguishable from the last iteration, making it impossible to predict the exit branch. With the VPC mechanism, each iteration uses a different VPC, allowing the predictor to distinguish each iteration from one another. In this example, the maximum value of the global backward counter is set to 4, forcing a history update after iteration 5, as shown in the example.

Note that the counter is cleared on every history update, not just updates from backward branches. This means that if any other branch in the loop body updates the history, the counter resets automatically, and the loop branch's VPC remains at PC + 0. The mechanism naturally avoids unnecessary forced updates without any special-case logic.

Not all predictor components benefit equally from VPC disambiguation. In TAGE, the bimodal table is indexed by PC alone and captures the overall taken/not-

Iteration	No Skipping (Baseline)		Naive Skipping		VPC Skipping	
	PC	**Hist**	**PC**	**Hist**	**VPC**	**Hist**
1	PC	H_1	PC	H	PC + 0	H
2	PC	H_2	PC	H	PC + 1	H
3	PC	H_3	PC	H	PC + 2	H
4	PC	H_4	PC	H	PC + 3	H
5	PC	H_5	PC	H	PC + 4	H
6	PC	H_6	PC	H	PC + 0	H_1
7	PC	H_7	PC	H	PC + 1	H_1
8	PC	H_8	PC	H	PC + 2	H_1
9	PC	H_9	PC	H	PC + 3	H_1
10	PC	H_{10}	PC	H	PC + 4	H_1

Table 3.1: Comparison of history update strategies for a 10-iteration loop.

taken tendency of a branch across all its dynamic instances. Applying the VPC would cause each dynamic instance to map to a different entry, fragmenting this estimate across iterations and preventing the table from accumulating sufficient observations to make accurate predictions. Instead, the bimodal table is always indexed with the original PC, while all history-based tables use the VPC together with the specified history length.

3.3.2 Direct Unconditional Branches

Direct unconditional branches never introduce a new control flow pattern because they unconditionally jump to a statically determined target. As a result, they do not carry any information about the control flow, and can be safely skipped on history update without decreasing prediction accuracy.

3.3.2.1 Calls and Returns

Calls always update the history register, encoding the call site context into the global history. Returns, despite having data-dependent targets, are unconditionally skipped from history updates. Since the return target is fully determined by the

call site, and the call is already present in the history, the return cannot introduce a new control-flow path beyond what the call has already established. By the same argument as predictable branches, the return's history update is strictly redundant and can be safely omitted.

3.3.2.2 Other Indirect Branches

Indirect branches, where the branch target depends on either register or memory values which varies at runtime, are handled using ITTAGE [**?**], the indirect branch counterpart to TAGE. Since ITTAGE is also based on history-to-target correlations, the same skipping policy applies directly — a predictable indirect branch does not introduce new control-flow information and can be excluded from history updates under the same criteria as direct branches.

3.3.2.3 Branches That Depend on Deep History

For branches that are only predictable with a long history, although updating the history register does not add new information, it can still provide benefits by effectively bringing the relevant correlation to the front of the history register and preventing it from being pushed out. Consider the following example in Listing 3.3, assuming the history register can only hold 64 updates. The question is whether BR3 should skip its history update.

```
while(1){
    if (A) { ... }              // BR1: unpredictable
    for (i = 1 to 50) {
        if (random()) ...       // BR2: unpredictable
    }
    if (A) { ... }              // BR3: predictable, correlated with BR1
    for (i = 1 to 50) {
        if (random()) ...       // BR4: unpredictable
    }
    if (A) { ... }              // BR5: depends on BR1
}
```

Listing 3.3: Branch predictability dependent on history depth

BR3 is predictable because its outcome correlates with BR1, which is already in the history register. Since BR3 is predictable, a naive skipping policy would exclude it from history updates. However, consider how long BR1's information remains alive in the history register under each case. Without skipping BR3, BR1's information is refreshed to the front of the history register each time BR3 updates, keeping it alive for the full duration of the loop — it remains within the 64-entry window when BR5 is reached. If BR3's update is skipped, BR1's information is never refreshed, and the 50 subsequent instances of BR4 push it out of the window entirely before BR5 is reached, making BR5 unpredictable.

While BR5 illustrates the concrete consequence of skipping BR3, the skipping decision for BR3 cannot and should not depend on the existence of BR5 — there is no way to know in advance which downstream branches may depend on BR1's correlation. The right criterion is simpler: if BR3 depends on deep history that is at risk of being pushed out, it should update the history register regardless, keeping that correlation alive for longer.

To identify such cases, I use the TAGE table that produces a tag match as a proxy for history depth dependency. Since each TAGE table is associated with a specific history length, a tag match in a long-history table indicates that the branch's behavior correlates with a deep history pattern — one that is at risk of being pushed out. If a branch frequently receives a tag match from a table whose history length exceeds a predetermined threshold, its history update is retained to preserve that correlation.

3.3.2.4 Supporting Branches that Are Only Predictable under Some Control Flows

Some branches are only predictable under specific control-flow contexts. To track such cases, any implementation must rely on some representation of the current control-flow context to determine whether a branch should skip a history update. However, once skipping begins, that representation diverges from what was seen dur-

ing training — the skipped updates mean the predictor encounters a different history state than the one under which it learned to identify skippable branches. Without a stable reference that remains consistent between training and prediction, the predictor can no longer reliably recognize the same control-flow pattern and correctly identify which branches should be skipped.

Fortunately, TAGE maintains two forms of history: the global history register (GHR) and the path history register. The GHR is much longer (hundreds to thousands of bits) and captures deep control flow context, while the path history is shorter (tens of bits) and tracks only recent control-flow information. By skipping updates only in the GHR while always updating the path history, the predictor preserves a stable representation of control flow for identifying branches that are predictable under certain contexts. This approach allows the GHR to effectively extend its usable length without losing the ability to track branches that require context, enabling deeper correlations to be captured safely even when skipping updates.

3.3.3 The Effects of Different Skipping Policies

The goal of this study is to understand how different pruning policies would affect prediction accuracy independent of any particular tracking mechanism. To achieve this, each trace is run twice. During the first run, the prediction accuracy of each branch is recorded, along with the percentage of time each branch receives tag matches in TAGE from a long-history table. Branches are identified as eligible for skipping if their prediction accuracy exceeds a predetermined threshold and they frequently only receive tag matches from short-history tables — indicating that they do not depend on deep history correlations that are at risk of displacement. During the second run, branches identified as eligible are excluded from GHR updates, with the VPC mechanism applied to handle backward branches as described in Section 3.3.2.2. Prediction accuracy is then measured under the resulting filtered history.

3.3.3.1 Methodology

I use the simulator provided by CBP6 [5] to evaluate the impact of skipping predictable branches on history quality. The baseline predictor is TAGE with a 256-bit history register. To isolate the effect of history skipping from capacity and aliasing effects, each TAGE tagged table is sized to be 16K entries instead of 1K entry and tag widths are increased from 8/12 bits to 27/31 bits to minimize false matches. I modified the history update algorithm so that 4 bits are pushed to the global history register per taken branch, matching the history update algorithm described in the TAGE cookbook [57].

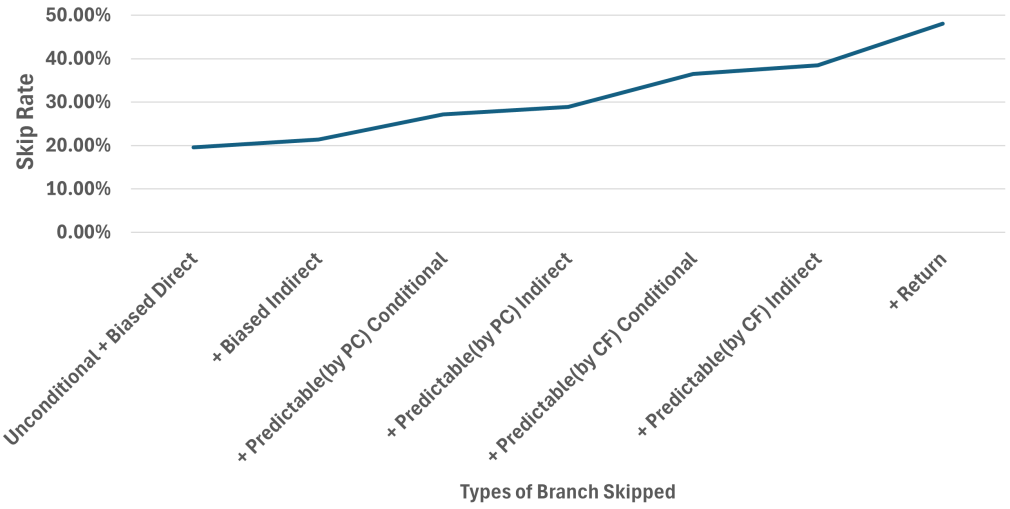

Figure 3.2: Skip Rate vs. Types of Branches Skipped

3.3.3.2 Overall Skip Rate and MPKI improvement

Figure 3.2 and Figure 3.3 show the skip rate and the absolute MPKI improvement over a baseline TAGE (with the same idealization as described in 3.3.3.1) of my history pruning mechanism. Each x-axis label represents a cumulative set of skipped branch types, where each successive label adds one more branch type to those already skipped. The results show that when all predictable branches are skipped, 48.1% of

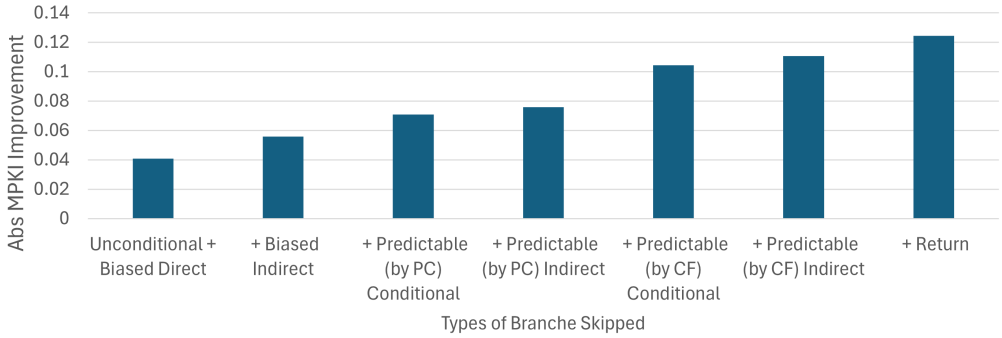

Figure 3.3: MPKI Improvement vs. Types of Branches Skipped

total updates are skipped, with some traces skipping as much as 91.1%. Since these updates add no new information to the history register, skipping them not only preserves prediction accuracy but improves the effective history length, resulting in an average improvement of 0.12 MPKI and a maximum improvement of 1.1 MPKI. In comparison, prior work only skips unconditional and biased direct branches, achieving a skip rate of only 19.6% and an MPKI improvement of only 0.04, less than one third of the improvement achieved by skipping all predictable branches.

3.3.3.3 Sensitivity to VPC Displacement Bound

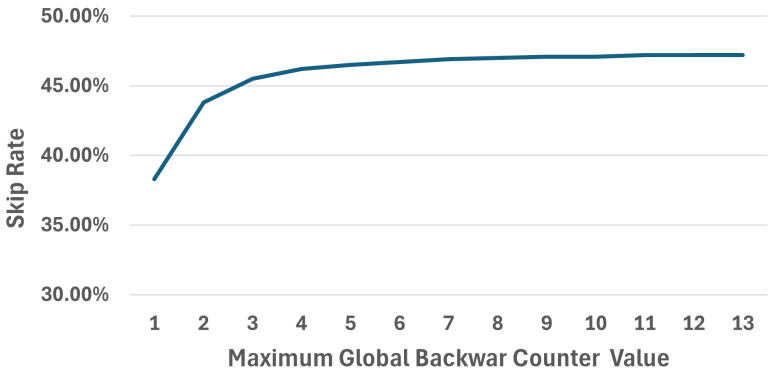

Figure 3.4: Skip Rate vs. VPC Displacement Upperbound

Figure 3.4 and Figure 3.5 show how skip rate and MPKI vary with different

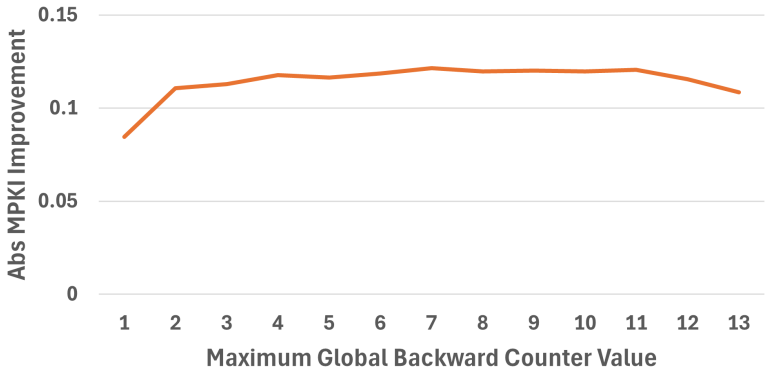

Figure 3.5: Abs. MPKI Improvement vs VPC Displacement Upperbound

maximum global backward counter values. As discussed in Section 3.3.1.1, the VPC is formed by combining the global backward counter with the original PC, where the maximum counter value determines how many times a PC can be incremented before a history update is forced. A larger maximum increases skipping opportunity but also raises the risk of VPC collision with another static branch's PC, introducing aliasing. The results show that allowing the PC to change just 3 times captures the majority of skipping benefit, with the skip rate plateauing shortly after and remaining nearly flat through 13. MPKI improvement follows a similar trend, rising sharply from 0.0846 at a counter value of 1 to 0.1128 by 2, before peaking at 0.1215 at a counter value of 7, beyond which aliasing begins to slightly erode the gains. This suggests that a maximum counter value of 7 represents the optimal tradeoff between skipping opportunity and aliasing.

3.3.3.4 Skipping Over Branches that Use Long Histories

The 2016 TAGE features 18 unique history lengths with 36 possible tagged tables. Table IDs range from 1 to 36, where higher IDs correspond to longer history lengths, and each history length is shared by two consecutive tables. Figure 3.6 and Figure 3.7 show how skip rate and MPKI vary with the threshold used to classify a table as long-history. A lower threshold classifies more tables as long-history, causing

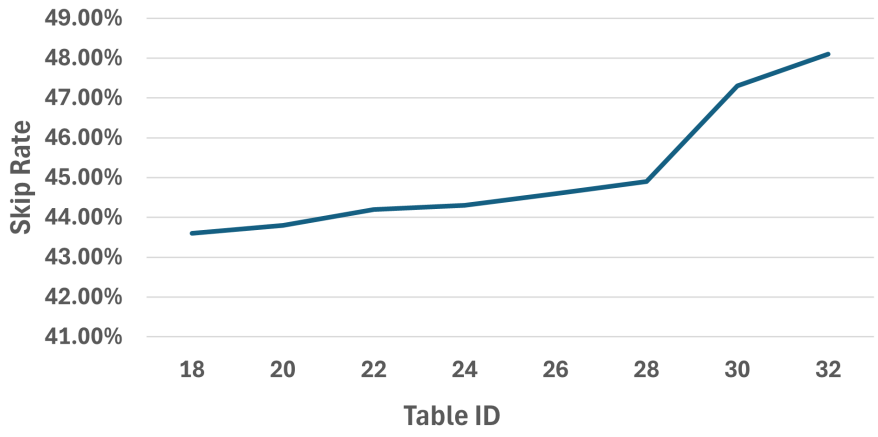

Figure 3.6: Skip Rate vs. First Long History Table ID

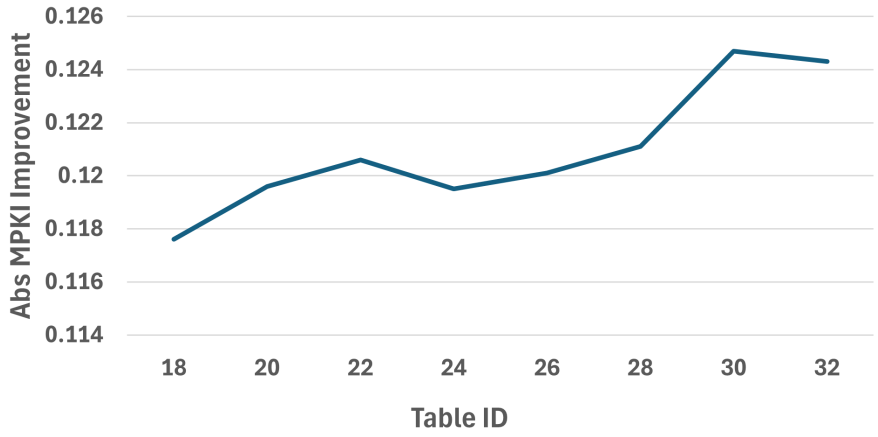

Figure 3.7: Abs. MPKI Improvement vs. First Long History Table ID

more branches to be ineligible due to deep history dependency, reducing the skip rate. A higher threshold classifies fewer tables as long-history, increasing the skip rate but risking the displacement of deep correlations that branches depend on. The results show that the optimal threshold for MPKI is table 30, which uses a 132-bit global history out of 256 total, beyond which the cost of losing deep history correlations outweighs the benefit of additional skipping.

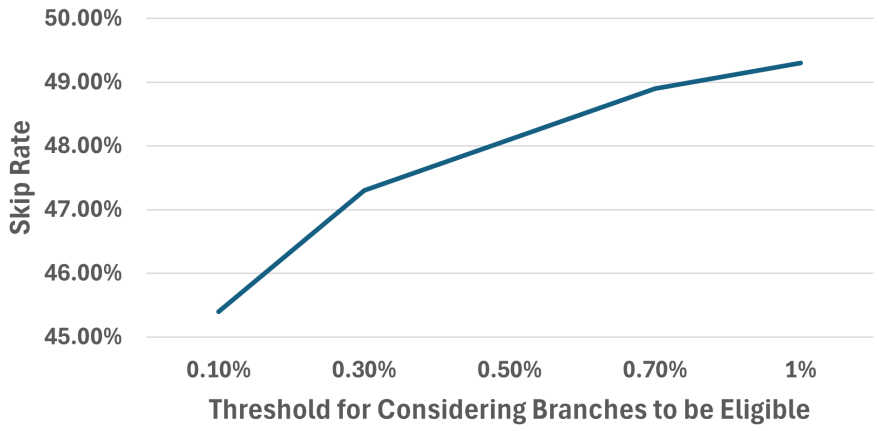

Figure 3.8: Skip Rate vs. First Long History Table ID

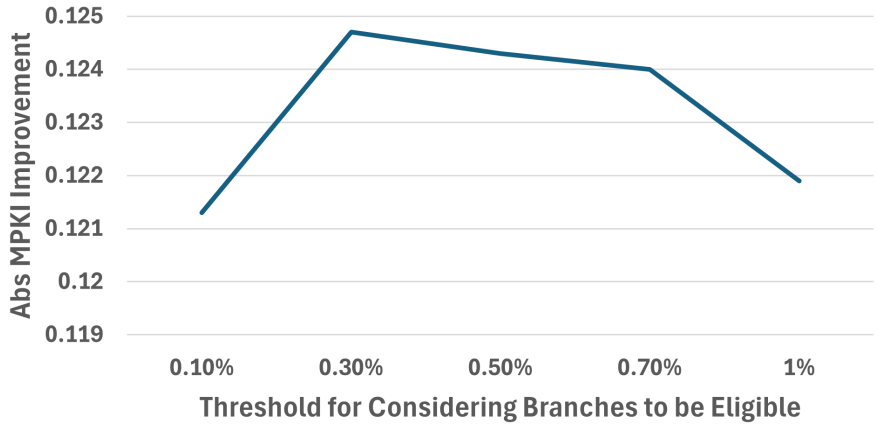

Figure 3.9: Abs. MPKI Improvement vs. First Long History Table ID

3.3.4 Ineligibility Event Rate Threshold

Figure 3.8 and Figure 3.9 show how skip rate and MPKI vary with the ineligibility event rate threshold. An ineligibility event is defined as either a misprediction or a prediction sourced from a long-history table. Only branches whose ineligibility event rate falls below the threshold are considered eligible for skipping. A higher threshold admits more branches as eligible, increasing the skip rate but also including branches that are either not reliably predictable or depend on deep history correla-

tions, degrading prediction accuracy. The results show that even at the strictest threshold of 0.1%, more than 45% of updates are skipped, demonstrating that a large fraction of branches in typical programs are highly predictable. The skip rate increases steadily to 49.3% at a threshold of 1%, however MPKI improvement peaks at 0.1247 at a threshold of 0.3% and degrades noticeably beyond that, confirming that the additional branches admitted at higher thresholds hurt more than they help. This makes 0.3% the optimal threshold, balancing skipping opportunity against prediction accuracy.

3.3.5 Sensitivity to PHIST Length used to Identify Predictable Branches

In the oracle experiment, predictable branches are identified using two criteria: PC alone, and (PHIST, PC) together. The PHIST length controls how many bits of path history are used in the latter identifier, determining how much context is used to classify a branch as predictable. A longer PHIST can identify branches that are only predictable under specific contexts, increasing the skip rate, but a branch that appears as a single predictable pattern under a short PHIST may fragment into many distinct contexts under a longer one, each seen too infrequently to confidently classify the branch as predictable. While this fragmentation does not affect oracle results since predictability is known before simulation starts, it poses a practical challenge for real hardware implementations that must learn predictability online.

Figure 3.10 and Figure 3.11 show how skip rate and MPKI vary with PHIST length. The skip rate grows steadily from 42.3% at 4 bits to 48.1% at 27 bits. However, MPKI improvement remains stable across all configurations, ranging between 0.10 and 0.126 with no clear monotonic trend, suggesting that PHIST length has little impact on prediction quality. Notably, even 4 bits achieves within 0.01 MPKI of the best result at 27 bits while only sacrificing 6% in skip rate coverage, making shorter PHIST lengths an attractive option during implementation.

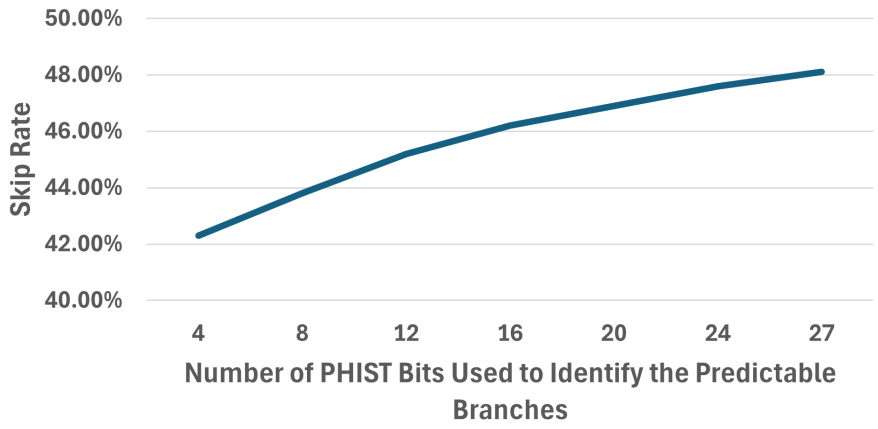

Figure 3.10: Skip Rate vs. PHIST Length Used

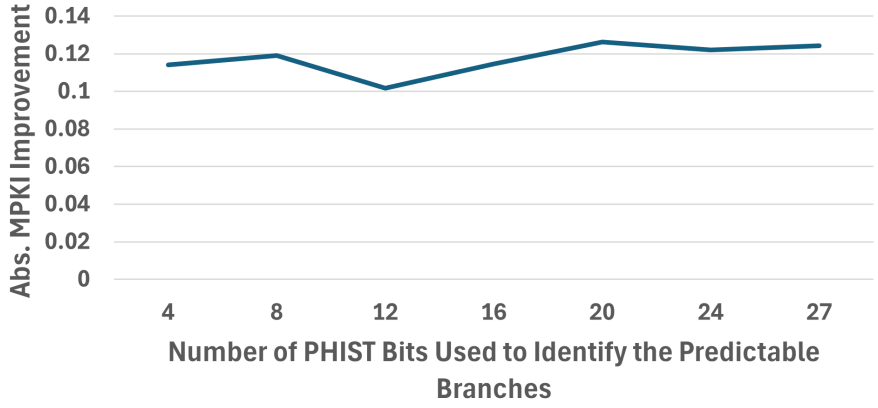

Figure 3.11: Abs. MPKI Improvement vs. PHIST Length Used

3.3.6 Per Trace Skip Rate and MPKI(S-Curves)

Figure 3.12 and Figure 3.13 show the per-trace MPKI improvement and skip rate of the best configuration, with traces sorted in descending order. The skip rate curve shows a smooth decline from a maximum of 91.1%, with the majority of traces falling in the 40-70% range, confirming that most programs contain a substantial fraction of predictable branches. The MPKI curve shows that the majority of traces see a positive improvement, with one trace benefiting by 1.1 MPKI, while only a

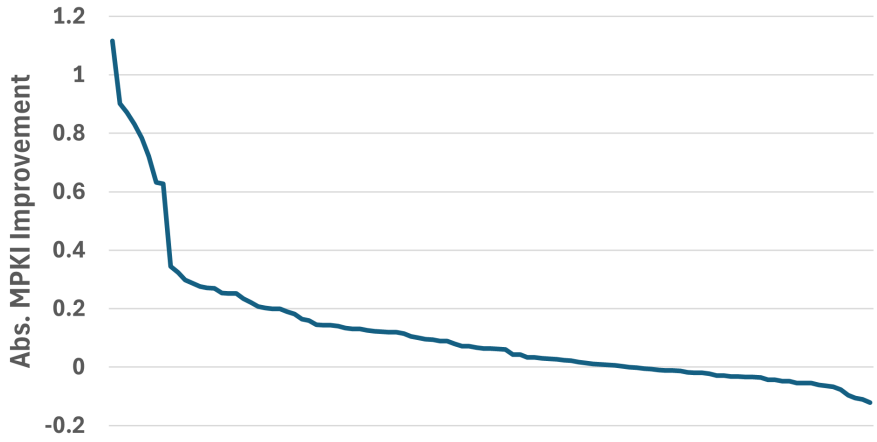

Figure 3.12: S-Curve for MPKI Improvement

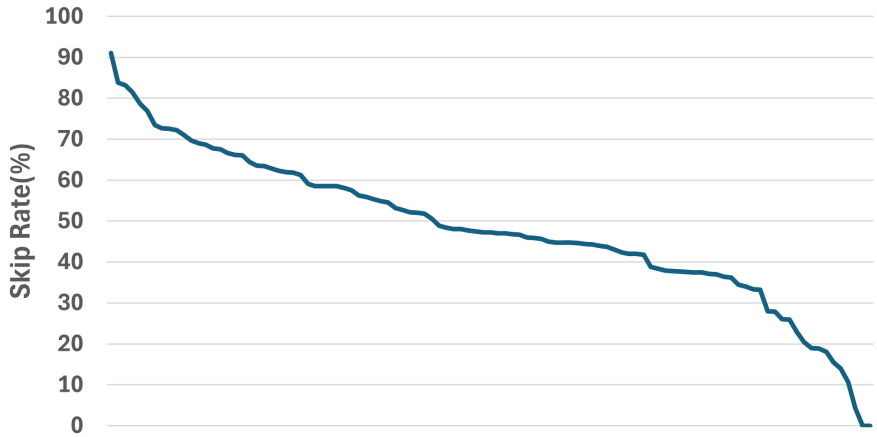

Figure 3.13: S-Curve for Skip Rate

small number of traces experience slight regressions of up to 0.2 MPKI. Overall, the mechanism is broadly beneficial across workloads, with the average improvement of 0.1242 MPKI driven by consistent gains across most traces rather than a few outliers.

Furthermore, restricting to the 71 traces where history length has a meaningful impact (at least 0.1 MPKI difference between a 256-bit and 3000-bit history), the average MPKI improvement increases to 0.1928, demonstrating that the mechanism is most effective precisely where history quality matters most.

3.3.7 Effect of History Pruning on Aliasing

To understand the effect of history pruning on aliasing, the following experiments were carried out to see the differences in MPKI with respect to different tag widths, both with history pruning and without history pruning.To isolate the effect of history pruning on aliasing, the history length is fixed at 3000 bits. This length is chosen as a saturation point, beyond which no further MPKI improvement is observed. Any remaining difference between pruned and un-pruned configurations therefore reflects the intrinsic cost of pruning, whether from increased aliasing or information loss. The same set of branches skips history updates regardless of the number of tag bits used here. Table 3.2 shows the MPKI results.

Tag Width	**8-bit**	**12-bit**	**16-bit**	**20-bit**	**8 to 20 Diff**
Not Pruned	3.6440	3.6152	3.6178	3.6183	−0.0257
Pruned	3.6912	3.6484	3.6526	3.6543	−0.0369

Table 3.2: MPKI at history length 3000 (aliasing isolation experiment)

As shown in Table 3.2, increasing the tag width from 8 to 20 bits yields a larger MPKI improvement under history pruning (−0.037) than without (−0.026). Since wider tags reduce aliasing by providing more precise branch identification, the greater benefit of wider tags under pruning suggests that history pruning introduces additional aliasing. This suggests that resizing the TAGE predictor with wider tags would potentially improve the MPKI when history pruning is enabled.

3.4 Implementation

3.4.1 Tracking the Predictable Branch

To determine whether a prediction packet should skip a history update, the predictor must track which packets are predictable and under what conditions. Some packets are predictable regardless of control-flow context, while others are only predictable under specific contexts. These two cases are tracked separately using two

cache-like structures.

The first structure, the *good PC table*, tracks prediction packets that are predictable regardless of control-flow context. It is indexed by prediction-packet PC, and each entry holds a saturating counter. When the last instruction in a prediction packet commits, the counter is incremented if no misprediction occurred, and it did not use a long history table within the packet, and none of the branches in the packet used long history. Otherwise, the counter is decremented. Counters are also decremented periodically to handle phase changes, ensuring that packets that stop showing up over time are eventually removed. If a packet attempts to increment its counter but no corresponding entry exists, I will try to allocate an entry for this packet by replacing the entry in the same set whose counter value is 0. If no such entry exists, all counters in the set are decremented by one, and the allocation request is dropped. A prediction packet is considered eligible for skipping if its counter exceeds a threshold.

The second structure, the *good CF table*, tracks prediction packets that are predictable only under specific control-flow contexts. It uses the same update, decay, and replacement policy as the good PC table, but is indexed by (path history, prediction-packet PC) rather than PC alone. The good CF table is only updated for a given PC if that PC is not already eligible in the good PC table, ensuring the two structures track complementary cases without redundancy.

Table 3.3 provides a summary of how the counters are updated in the two caches. A prediction packet is eligible to skip the history update after it reaches a predetermined threshold. I experimentally determine this to be 2024.

3.4.2 Locked Tables

Each time the set of skipped prediction packets changes, the global history seen by every branch changes as well, requiring the predictor to warm up again before it can predict accurately. If such changes happen frequently, the predictor spends most of its time warming up and never reaches peak accuracy. To prevent this, the predictor

Trigger	Event	Counter Adjustment
Every retired prediction packet	Misprediction	-180
	Tag match from long-history table (no misprediction)	-180
	Correct prediction, not from long-history table	$+1$
Every N instructions	Periodic decrement (full set)	-8
Allocation on Increment Miss	Failed allocation attempt	-1

Table 3.3: Counter Adjustment for Tracking Packet Eligibility

maintains two separate sets of tables: a *training table* that is updated continuously, and a *locked table* that determines which packets would skip the history update.

When a prediction packet becomes eligible to be skipped in the training table, it is not immediately added to the locked table. Instead, updates to the locked table are batched so that they will be applied once, causing only one warm up to the predictor. However, not every batch of changes is worth applying — if the difference between the locked table and the training table is small, the benefit of updating may not justify the warm-up cost. I define a divergence score which measures the difference between the two tables as follows:

```
A = # of entries eligible in training table but not in locked table
B = # of entries eligible in locked table but not in training table
M = Scaling Factor
divergence_score = A + M * B
```

Listing 3.4: Computation of divergence score

A represents new skipping opportunities — packets that have become predictable and could benefit from skipping. B represents packets that are currently being skipped but are no longer predictable in the training table, either because their behavior has changed or they have not been observed recently. Entries in B are weighted more heavily through the scaling factor M because retaining them in the

locked table risks introducing information loss into the history register, potentially degrading the prediction of downstream branches. The locked table is only updated when the divergence score exceeds a predetermined threshold, ensuring that updates are applied only when the expected benefit outweighs the warm-up cost.

3.4.3 Blacklist

Some prediction packets are intermittently predictable — they alternate between predictable and unpredictable phases, causing them to repeatedly enter and exit eligibility in the training table. Each such transition triggers a warm-up cost, and packets that oscillate frequently can dominate the warm-up overhead without providing sustained benefit. To exclude such cases, two small blacklists are introduced, one for the PC tables and one for the CF tables.

Each blacklist is a cache-like structure that tracks prediction packets that have demonstrated unstable predictability. An entry in each blacklist consists of a 3 bit counter. When a packet misprediction would cause its counter in the training table to drop below the eligibility threshold — indicating that the packet is transitioning from predictable to unpredictable, the packet's counter in the blacklist is incremented by 1. If the counter in the blacklist is above a threshold, the corresponding packet skips the training in the training tables entirely, ensuring that intermittently predictable packets cannot repeatedly trigger warm-up cycles.

Figure 3.14 shows the overall architecture between the training table, locked table, and the blacklist. During prediction time, the fetch PC is used to index the locked table; a hit would mean this current prediction packet should skip history update. When the last instruction in the fetch group retires, it would check if the entry exist in the black list. If the entry is in the blacklist, then the update is skipped. If an entry is not in the blacklist, it then proceeds to update the training table.

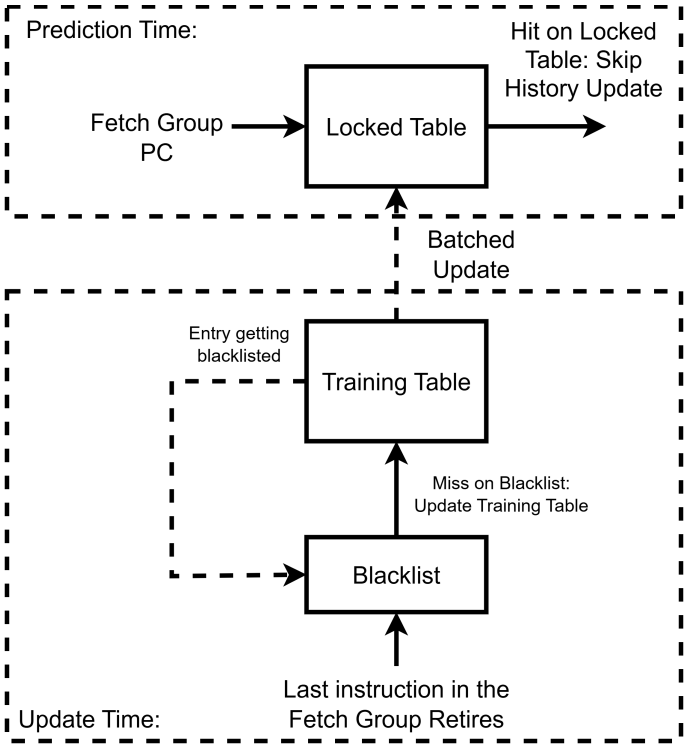

Figure 3.14: Overall Architecture

3.4.4 Hardware Overhead

The main hardware overhead is the 2 sets of good PC table and good CF table. The good PC training table holds 1024 entries(128 sets by 8 ways), each entry contains a 41-bit tag for PC and an 12-bit counter. The good CF training table also holds 1024 entries, each entry contains a 49-bit (41 bits for PC + 8 bits for phist) tag and an 12-bit counter. The locked tables have the same number of entries, but instead of a counter, they only hold a valid bit indicating if an entry is eligible. Therefore, each entry in the locked good PC table is 42 bits, and each entry in the locked good CF table is 50 bits. The total storage among the 4 tables is 25.75KB.

The PC blacklist and CF blacklist each have 8 entries(1 sets by 8 ways). The tag for the PC blacklist is 46 bits, and the tag for the CF blacklist is 54 bits, the same as above. Each entry holds a 3-bit counter. The total storage overhead for the

blacklist is 106B.

The global backward counter also needs to be checkpointed, adding 4 bits per checkpoint. Assuming a processor with a 1024 instruction window supporting up to 256 in-flight branches, this would require 1024 more bits (0.125KB) overall.

It should be noted that only 12.875KB of storage (the locked tables) is needed in the processor frontend, where the predictor must decide at fetch time whether a prediction packet should skip its history update. The remaining 13.103KB (training tables and blacklists) are only accessed at retirement and do not lie on any critical timing path, allowing them to be placed anywhere on chip without impacting performance.

3.5 Results

3.5.1 Methodology

I evaluate my branch history pruning algorithm and the runtime predictable branch detection mechanism on the CBP6 [5] simulation infrastructure. The baseline predictor is a TAGE predictor from the TAGE-SC-L [56] predictor from CBP5 [1] with 2 modifications. First, the history update is modified so that one update is applied for each taken branch, as specified by TAGE Cookbook [57]. Secondly, the size of the global history register is reduced to 256 bits to represent a realistic implementation.

The MPKI and skip rate reported in this section follow the same rule as CBP6 [5], which uses the first half of the trace as warm-up, and the second half of the trace for reporting.

3.5.2 MPKI and Skip Rate

Figure 3.15 and Figure 3.16 show the skip rate and MPKI improvement per benchmark category. FP and INFRA traces benefit the most from history pruning, as these workloads are the most sensitive to history length. Both categories contain

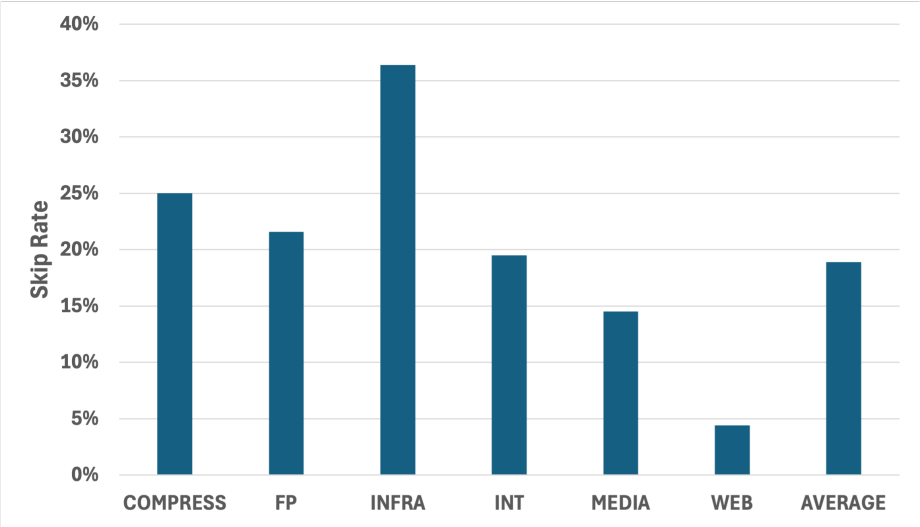

Figure 3.15: Skip Rate across Different Trace Categories

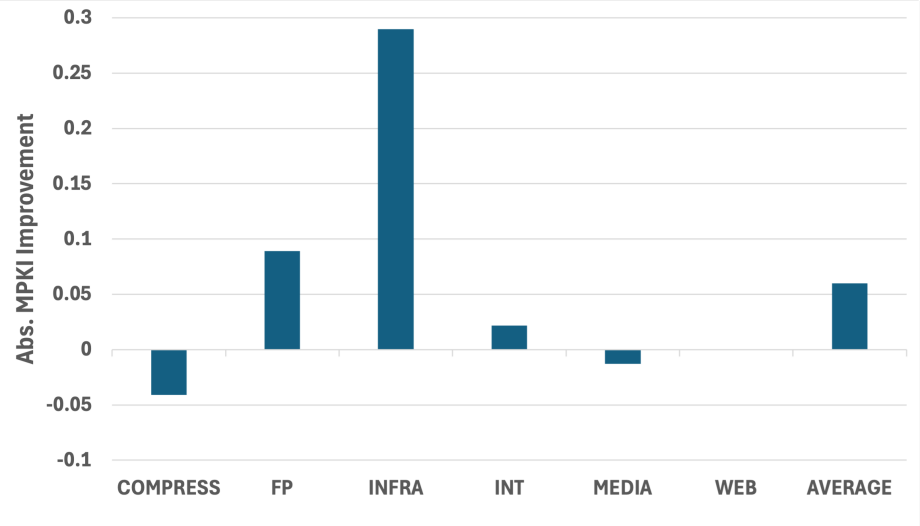

Figure 3.16: Abs. MPKI Improvement across Different Trace Categories

a high proportion of predictable branches — more than 21% of history updates are skipped in FP traces (up to 82%), and 36% are skipped in INFRA traces (up to 87%), providing ample opportunity for history compression. COMPRESS, despite skipping 25% of history updates, shows no MPKI improvement because the majority of these traces are not sensitive to history length, exhibiting less than 0.1 MPKI

difference between 256 bits and 3000 bits of history. Only 4% of updates are skipped in WEB traces, indicating that there are fewer highly predictable packets in them. Many INT traces suffer from frequent phase changes, making it harder for the tracking mechanism to identify and consistently skip predictable branches at runtime, resulting in only modest MPKI improvement. Overall, the mechanism skips 18.9% of packet updates on average and achieves a 0.0625 MPKI improvement.

To better isolate the benefit of history pruning, restricting the analysis to the 71 traces out of 105 that are sensitive to history length — defined as those exhibiting more than 0.1 MPKI difference between 256 bits and 3000 bits of history — the mechanism skips 19.1% of packet updates on average and achieves a 0.0977 MPKI improvement. This suggests that the overall average is diluted by traces that are inherently insensitive to history length, and that the mechanism delivers meaningful gains precisely where longer history matters most.

A baseline predictor with a 20% longer history register (320 bits) achieves only 0.0428 MPKI improvement, far less than the 0.0625 MPKI reduction delivered by my history pruning mechanism. The pruning is more aggressive on traces that are sensitive to history sizes such as INFRA traces, where my mechanism achieves a 0.29 MPKI reduction compared to the 0.07 MPKI reduction from increasing the history register size by 20%.

3.5.3 Size of PHIST Used to Identify Predictable Branches

Recall that the PHIST length controls how much path history context is used in the (PHIST, PC) identifier for the good cf table when classifying a branch as predictable. Figure 3.17 and Figure 3.18 show the skip rate and MPKI with respect to the number of path history bits used to identify predictable branches. Interestingly, unlike the oracle experiment in Section 3.3.5, the skip rate does not increase monotonically with the number of path history bits. The oracle experiment is immune to this effect because predictability is determined offline with full knowledge of all branch

instances — coverage is never an issue. In the runtime implementation, however, the good CF table must learn online from limited observations. While longer path history allows more branches to be identified as predictable, it also means each specific (PC, path history) combination is seen less frequently, reducing the confidence accumulated per entry and making it harder to cross the eligibility threshold. This mirrors the fundamental coverage vs. precision tradeoff in TAGE itself, where longer history tables have better precision but worse coverage because entries are harder to train. The results show that 8 bits of path history provides the best tradeoff, maximizing both skip rate and prediction accuracy.

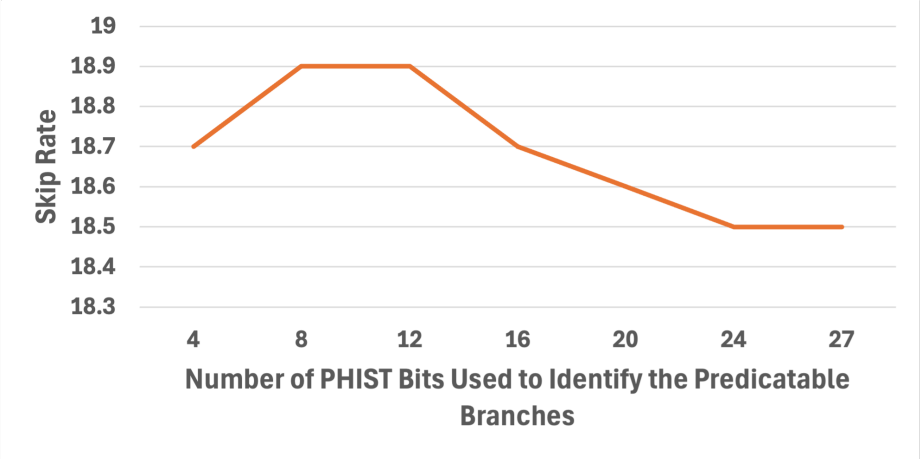

Figure 3.17: Skip Rate vs. Number of PHIST Length Used

3.5.4 Divergence Score

Figure 3.19 shows how MPKI varies with both the divergence score threshold and the multiplying factor as described in Listing 3.4. A lower divergence score threshold provides more opportunity to adapt to phase changes, as updates from the training table are applied to the locked table more frequently, but comes at the cost of more frequent warm-ups. The multiplying factor controls how much weight is given to entries that were previously eligible but are no longer predictable, relative to entries that have newly become predictable. A higher multiplying factor makes the

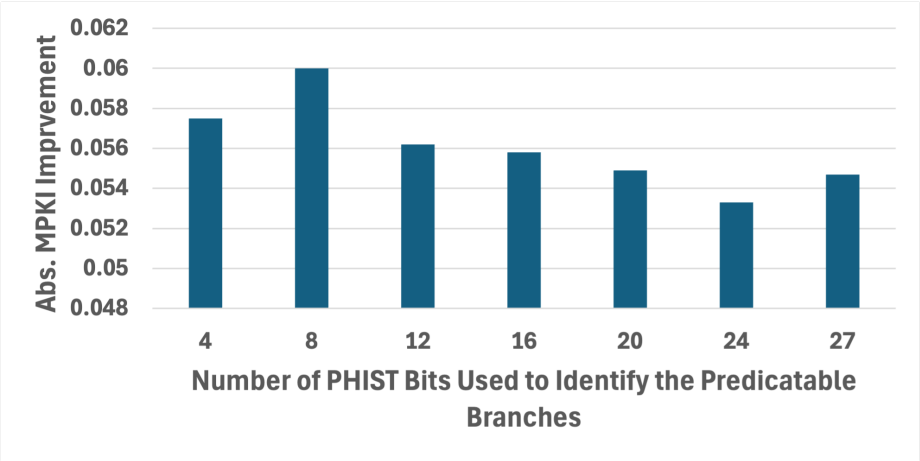

Figure 3.18: Abs. MPKI Improvement vs. PHIST Length Used

predictor more conservative about applying updates that remove previously skipped branches, since doing so risks introducing information loss into the history register. The results show that a divergence score threshold of 185 with a multiplying factor of 2 provides the best balance between phase adaptation and warm-up cost, yielding the most MPKI improvement. Notably, 87 out of the 105 traces triggered only a single locked table update during the entire simulation, demonstrating that an unstable view of history can drastically reduce the performance benefit of a longer effective history register.

3.6 Discussion on Skip Rate

While the oracle experiment achieves a 48.1% skip rate on average, our practical implementation achieves only 19.1%. The primary reason for this gap is the divergence score threshold that controls when the locked table is updated. Even when many packets have accumulated sufficient evidence of predictability in the training table, the locked table is only updated when the divergence score exceeds a predetermined threshold. The threshold is chosen to optimize MPKI with infrequent updates to the locked table, keeping warm-up costs low at the expense of a lower skip rate.

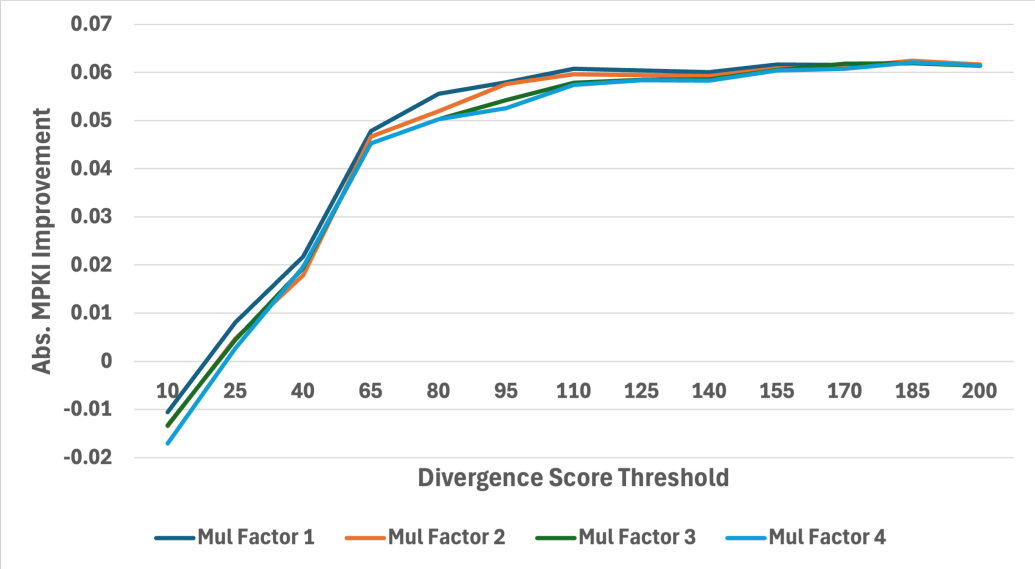

Figure 3.19: Abs. MPKI Improvement vs Divergence Score Threshold at Different Multiplying Factors

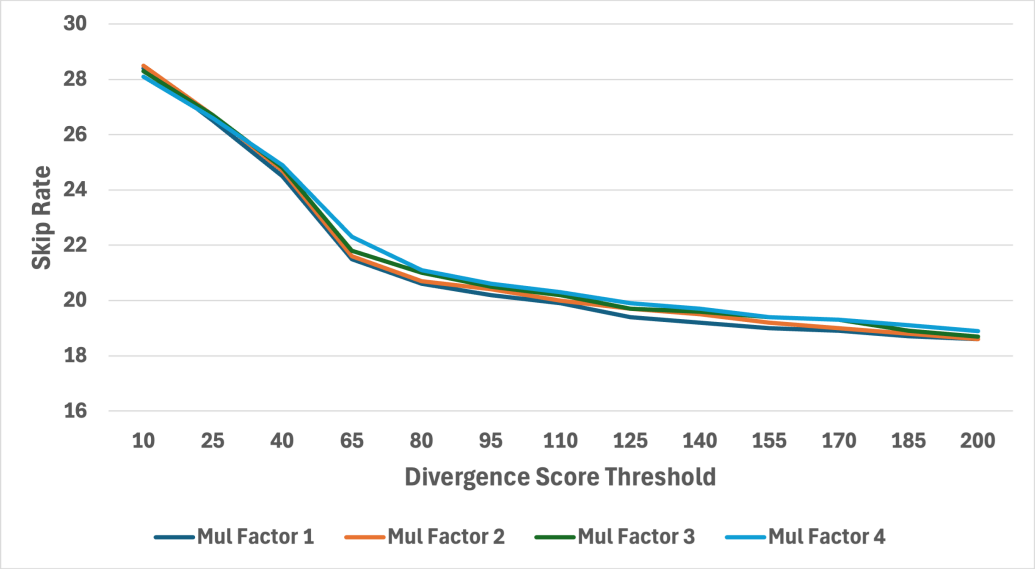

Figure 3.20: Skip Rate vs Divergence Score Threshold at Different Multiplying Factors

As a result, the dynamic tracking algorithm is cannot adapt to phase changes very well, resulting in lower skip rate compared to an oracle implementation.

Figure3.20 shows the skip rate of different divergence score thresholds. Low-

ering the divergence score from 185 to 10 would increase the skip rate to 28.5% but would also increase MPKI from the excessive warm-up cost introduced by frequent updates to the locked table. On top of a lowered threshold(1024 as opposed to 2014) would allow branches that show up infrequently to be captured, resulting in 32.2% skip rate.

3.7 Related Work

3.7.1 History Filtering for a Specific Branch

Prior work [67, 68] has shown that a particular branch only depends on a few branches in the global history. Their primary motivation was to reduce the exponential storage cost that arises when uncorrelated branches pollute the history: by learning an offline per-branch filter that retains only the relevant correlating branches, they avoid the combinatorial explosion of history patterns. A side effect of this filtering, however, is that the history now covers a longer effective region, since it is only updated by the small subset of branches that actually matter for each prediction. While my goal is different — I aim to extend effective history length directly by skipping updates from unconditionally predictable branches — both approaches share the observation that the effective history length can be improved by being selective about which updates are included. Unlike their work, however, I do not claim that certain branches are irrelevant to all predictions; rather, I identify branches whose outcomes are already known and therefore contribute no new information to the history regardless of context.

3.7.2 Modulo Based History Filtering

Jiménez [26] proposes modulo history (MODHIST), where only branches whose PC is divisible by some fixed modulus are recorded in the history register. The primary motivation is to reduce aliasing pressure and training time in perceptron-based predictors — uncorrelated branches produce different hash values for histories that

should be treated identically, slowing convergence. While both MODHIST and my work filter branches from the history register, the goals and criteria are fundamentally different. MODHIST uses an arbitrary PC-based criterion with no regard for whether a filtered branch actually contributes useful information — as Jiménez himself acknowledges, filtered branches may carry genuine correlation that is simply discarded. My work instead targets a different problem entirely: extending the effective history length by removing branches whose outcomes are already determined by the existing history, ensuring that no genuine correlation is ever lost.

3.8 Conclusion

This chapter presented a branch history pruning mechanism that improves effective history length by skipping history updates from predictable branches. Since predictable branches contribute no new control-flow information, their history updates are strictly redundant, and skipping them allows the predictor to capture deeper correlations within the same physical history register.

The oracle study confirmed that up to 48.1% of history updates can be safely skipped, yielding an average MPKI improvement of 0.1242 and up to 1.1 MPKI on individual traces. On traces where history length has a meaningful impact, the improvement increases to 0.1928 MPKI. A run time implementation with only 26KB storage overhead achieves 0.06 MPKI reduction over the baseline TAGE predictor.

Chapter 4: Conclusion and Future Work

4.1 Conclusion

This dissertation challenges a foundational assumption in modern branch predictor design: that every branch outcome contributes new control-flow information. I have shown that this assumption is unnecessarily conservative — the majority of dynamic branches are predictable under a given control flow, and predictable branches do not introduce new control-flow paths. This insight forms the basis for two contributions that address two longstanding challenges in branch prediction: predictor latency and history length. The first contribution applies this insight to ahead prediction. Prior approaches to ahead prediction generate predictions for all 2^N possible missing history patterns when predicting N branches ahead, causing energy to grow exponentially with ahead distance. Because most branches are predictable, however, the vast majority of these patterns never materialize at runtime. By explicitly tracking only the missing history patterns that actually occur in practice, the proposed ahead predictor reduces per-prediction energy from 14.6× to 1.5× over the baseline, making ahead prediction practical within realistic energy constraints.

The second contribution applies this insight to history length. Because predictable branches do not introduce new control-flow information, recording them in the history register is redundant — any downstream branch that correlates with a predictable branch can correlate with the existing history directly. I propose a lossless branch history pruning algorithm by selectively omitting predictable branches from history updates, allowing the history register to capture a longer window of meaningful branch outcomes without physically increasing its length, reducing the storage, recovery latency, and energy costs that make long histories impractical in real processors.

Together, these two contributions demonstrate that the full global history is

not necessary for accurate branch prediction. Exploiting the predictable branches — rather than treating all branches as equally informative — enables more efficient predictors along multiple dimensions simultaneously.

4.2 Future Work

There are several ways to extend this thesis in the future. I will break them down into ahead prediction related and history pruning related.

4.2.1 Future Work on Ahead Prediction

- **Dynamically adjusting entries read per table.** While this dissertation shows that reading 2^N entries per table is unnecessary in most cases, it may be beneficial to dynamically increase the number of entries read when the number of missing history patterns is large. A mechanism that adapts dynamic behavior based on the number of primary tag hits could improve accuracy in hard-to-predict workloads without sacrificing energy efficiency in the easy ones.

- **Using ahead prediction to improve prediction throughput.** Supporting wider pipelines requires not just accurate predictions but higher prediction throughput — the ability to deliver multiple predictions per cycle. Ahead prediction is a natural fit for this problem, as it decouples the prediction of future branches from the resolution of the current one. Exploring ahead prediction as a mechanism for sustaining prediction throughput in wide-issue processors is a promising direction.

- **Ahead pipelining target prediction.** This dissertation focuses on ahead prediction of branch direction. Extending the ahead prediction framework to also predict branch targets ahead of time could further reduce front-end latency and improve prediction throughput.

4.2.2 Future work on History Pruning

- **Offline training.** The current mechanism identifies predictable branches dynamically at runtime. An offline training pass over representative workloads could produce a static skip list that is loaded into the predictor at run time, reducing cost associdated with having to track predictable branches dynamically, including warm up cost and extra storage.

- **Increasing prediction throughput.** The number of branches that can be predicted per cycle is fundamentally limited by how frequently the history register must be updated. Each history update changes the index and tag for every TAGE table[1], requiring a new lookup before the next prediction can be made. In the current design, this means prediction throughput is bounded by one taken branch per cycle, since every taken branch triggers a history update. However, if predictable branches do not update the history register, the index remains stable across consecutive predictable branches, allowing all of them to be predicted in a single lookup without serialization. This suggests that history pruning could extend the prediction packet to span all branches up to the first hard-to-predict branch, rather than just up to the first taken branch. Since most branches are predictable, this could substantially increase prediction throughput and better support wider pipelines.

- **Applying history pruning to other microarchitectural structures.** Branch history is used not only in branch predictors but also in other microarchitectural structures. For example, perceptron-based prefetch filtering [**?**] applies branch prediction techniques directly to prefetching decisions, demonstrating that branch history is a useful signal beyond branch prediction itself. Applying the same pruning principle to such structures — removing redundant history

[1] or any other history based predictor design

updates that contribute no new information — could yield similar benefits in terms of effective history coverage and prediction quality.

Works Cited

[1] Championship Branch Prediction. https://jilp.org/cbp2016/.

[2] Loongson 3A6000: A Star among Chinese CPUs. https://chipsandcheese.com/2024/03/13/loongson-3a6000-a-star-among-chinese-cpus/.

[3] Scarab. https://github.com/hpsresearchgroup/scarab.

[4] The standard performance evaluation corporation (spec), 2017. URL https://www.spec.org/cpu2017/.

[5] Cbp2025 simulator framework. https://ericrotenberg.wordpress.ncsu.edu/cbp2025-simulator-framework/, 2025. Accessed: 2026-02-06.

[6] Emet Behrendt, Shing Wai Pun, and Prashant J. Nair. Taming wild branches: Overcoming hard-to-predict branches using the bullseye predictor. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[7] Lingzhe (Chester) Cai, Aniket Deshmukh, Evan Lai, Ali Mansoorshahi, Mircea Tatulescu, Isaac Nudelman, and Yale Patt. Tage-sc-l with a code structure correlator. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[8] Daniel Chaver, Luis Piñuel, Manuel Prieto, Francisco Tirado, and Michael C. Huang. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ISLPED '03, page 390–395, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 158113682X. doi: 10.1145/871506.871603. URL https://doi.org/10.1145/871506.871603.

[9] Jian Chen and Lizy K. John. Autocorrelation analysis: A new and improved method for branch predictability characterization. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 194–203, 2011. doi: 10.1109/IISWC.2011.6114179.

[10] Aniket Deshmukh and Yale N. Patt. Criticality driven fetch. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 380–391, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480115. URL https://doi.org/10.1145/3466752.3480115.

[11] Aniket Deshmukh, Ruihao Li, Rathijit Sen, Robert R. Henry, Monica Beckwith, and Gagan Gupta. Performance characterization of .net benchmarks. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 107–117, 2021. doi: 10.1109/ISPASS51385.2021.00028.

[12] Aniket Deshmukh, Lingzhe Chester Cai, and Yale N. Patt. Alternate path fetch. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 1217–1229, 2024. doi: 10.1109/ISCA59077.2024.00091.

[13] Aniket Deshmukh, LingzheChester Cai, and Yale N. Patt. Timely, efficient, and accurate branch precomputation. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 480–492, 2024. doi: 10.1109/MICRO61859.2024.00043.

[14] M. Evers, S.J. Patel, R.S. Chappell, and Y.N. Patt. An analysis of correlation and predictability: what makes two-level branch predictors work. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*, pages 52–61, 1998. doi: 10.1109/ISCA.1998.694762.

[15] Jun Fan. Branch prediction via load value prediction: A case of ball (branch-alu-load-load) predictor. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[16] Dibakar Gope and Mikko H. Lipasti. Bias-free branch predictor. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 521–532, Dec 2014. doi: 10.1109/MICRO.2014.32.

[17] 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. doi: 10.1109/ISCA45697.2020.00015.

[18] Yasuo Ishii. Fused two-level branch prediction with ahead calculation. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9:1–19, 2007.

[19] D.A. Jimenez. Fast path-based neural branch prediction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 243–252, 2003. doi: 10.1109/MICRO.2003.1253199.

[20] D.A. Jimenez. Piecewise linear branch prediction. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 382–393, 2005. doi: 10.1109/ISCA.2005.40.

[21] D.A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, 2001. doi: 10.1109/HPCA.2001.903263.

[22] D.A. Jimenez, S.W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings 33rd Annual IEEE/ACM International*

Symposium on Microarchitecture. MICRO-33 2000, pages 67–76, 2000. doi: 10.1109/MICRO.2000.898059.

[23] Daniel A. Jiménez. Reconsidering complex branch predictors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, page 43, USA, 2003. IEEE Computer Society. ISBN 0769518710.

[24] Daniel A. Jiménez. Improved latency and accuracy for neural branch prediction. *ACM Trans. Comput. Syst.*, 23(2):197–218, may 2005. ISSN 0734-2071. doi: 10.1145/1062247.1062250. URL `https://doi.org/10.1145/1062247.1062250`.

[25] Daniel A. Jiménez. Multiperspective perceptron predictor. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[26] Daniel A. Jim´enez. Multiperspective perceptron predictor with tage. 2016. URL `https://api.semanticscholar.org/CorpusID:221213615`.

[27] Cansu Kaynak, Boris Grot, and Babak Falsafi. Confluence: Unified instruction supply for scale-out servers. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 166–177, 2015. doi: 10.1145/2830772.2830785.

[28] 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*, MICRO '21, page 816–829, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572. doi: 10.1145/3466752.3480124. URL `https://doi.org/10.1145/3466752.3480124`.

[29] 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, 2022. doi: 10.1109/MICRO56248.2022.00017.

[30] Hyesoon Kim, Jose A. Joao, Onur Mutlu, Chang Joo Lee, Yale N. Patt, and Robert 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. doi: 10.1109/TC.2008.227.

[31] Toru Koizumi, Toshiki Maekawa, Masanari Mizuno, Maru Kuroki, Tomoaki Tsumura, and Ryota Shioya. Runlts: Register-value-aware predictor utilizing nested large tables. *The 6th Championship Branch Prediction*, 2025.

[32] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. Blasting through the front-end bottleneck with shotgun. *SIGPLAN Not.*, 53(2):30–42, mar 2018. ISSN 0362-1340. doi: 10.1145/3296957.3173178. URL `https://doi.org/10.1145/3296957.3173178`.

[33] G.H. Loh. Revisiting the performance impact of branch predictor latencies. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 59–69, 2006. doi: 10.1109/ISPASS.2006.1620790.

[34] Yang Man, Lingrui Gou, Yuhang Liu, Mingyu Chen, and Yungang Bao. Lvcp: A load value correlated predictor for tage-sc-l. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[35] Scott Mcfarling. Combining branch predictors. 10 1998.

[36] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *1999 International Confer-*

ence on Parallel Architectures and Compilation Techniques (Cat. No.PR00425), pages 2–10, 1999. doi: 10.1109/PACT.1999.807388.

[37] Pierre Michaud, André Seznec, Stéphan Jourdan, and Pascal Sainrat. Alternative Schemes for High-Bandwidth Instruction Fetching. Research Report RR-3392, INRIA, 1998. URL `https://inria.hal.science/inria-00073297`.

[38] Milad Mohammadi, Song Han, Ehsan Atoofian, Amirali Baniasadi, Tor M. Aamodt, and William J. Dally. Energy efficient on-demand dynamic branch prediction models. *IEEE Transactions on Computers*, 69(3):453–465, 2020. doi: 10.1109/TC.2019.2956710.

[39] Karl H. Mose, Alexandra W. Chadwick, Marton Erdos, Jiayi Nie, Rika Antonova, Timothy M. Jones, and Robert D. Mullins. Pip: An ensemble of programming-idiom predictors. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[40] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 3–14, 2007. doi: 10.1109/MICRO.2007.33.

[41] Ajeya Naithani, Josué Feliu, Almutaz Adileh, and Lieven Eeckhout. Precise runahead execution. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 397–410, 2020. doi: 10.1109/HPCA47549.2020.00040.

[42] D. Parikh, K. Skadron, Yan Zhang, M. Barcella, and M.R. Stan. Power issues related to branch prediction. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 233–244, 2002. doi: 10.1109/HPCA.2002.995713.

[43] Y. N. Patt, W. M. Hwu, and M. Shebanow. Hps, a new microarchitecture: rationale and introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, page 103–108, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911725. doi: 10.1145/18927.18916. URL https://doi.org/10.1145/18927.18916.

[44] Y. N. Patt, S. W. Melvin, W. M. Hwu, and M. C. Shebanow. Critical issues regarding hps, a high performance microarchitecture. In *Proceedings of the 18th Annual Workshop on Microprogramming*, MICRO 18, page 109–116, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911725. doi: 10.1145/18927.18917. URL https://doi.org/10.1145/18927.18917.

[45] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. The arm neoverse n1 platform: Building blocks for the next-gen cloud-to-edge infrastructure soc. *IEEE Micro*, 40(2):53–62, 2020. doi: 10.1109/MM.2020.2972222.

[46] Arthur Perais and Rami Sheikh. Branch target buffer organizations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 240–253, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3623774. URL https://doi.org/10.1145/3613424.3623774.

[47] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 16–27, 1999. doi: 10.1109/MICRO.1999.809439.

[48] Alberto Ros. A deep dive into tage-sc-l. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025.

[49] David Schall, Andreas Sandberg, and Boris Grot. The last-level branch predictor. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 464–479, 2024. doi: 10.1109/MICRO61859.2024.00042.

[50] David Schall, Mária Ďuračková, and Boris Grot. The last-level branch predictor revisited. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–16, 2026. doi: 10.1109/HPCA68181.2026.11408567.

[51] David J. Schlais and Mikko H. Lipasti. Badgr: A practical ghr implementation for tage branch predictors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 536–543, 2016. doi: 10.1109/ICCD.2016.7753338.

[52] A. Seznec. Analysis of the o-geometric history length branch predictor. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 394–405, 2005. doi: 10.1109/ISCA.2005.13.

[53] A. Seznec and A. Fraboulet. Effective ahead pipelining of instruction block address generation. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pages 241–252, 2003. doi: 10.1109/ISCA.2003.1207004.

[54] André Seznec. The l-tage branch predictor. *The Journal of Instruction-Level Parallelism*, 9, May 2007. URL `https://jilp.org/vol9/v9paper6.pdf`.

[55] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9:1–6, 2007.

[56] André Seznec. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Pre-*

diction (CBP-5), Seoul, South Korea, June 2016. URL https://inria.hal.science/hal-01354253.

[57] André Seznec. TAGE: an engineering cookbook. Technical Report 9561, Inria, November 2024. URL https://hal.science/hal-04804900.

[58] André Seznec. Tage-sc for cbp2025. In *6th Championship Branch Prediction Workshop (in conjunction with ISCA 2025)*, Tokyo, Japan, 2025. URL https://ericrotenberg.wordpress.ncsu.edu/files/2025/06/cbp2025-final37-Seznec.pdf.

[59] André Seznec and Pierre Michaud. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism*, 8:23, February 2006. URL https://inria.hal.science/hal-03408381.

[60] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. Multiple-block ahead branch predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, page 116–127, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917677. doi: 10.1145/237090.237169. URL https://doi.org/10.1145/237090.237169.

[61] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. ASPLOS X, page 45–57, 2002. ISBN 1581135742. doi: 10.1145/605397.605403. URL https://doi.org/10.1145/605397.605403.

[62] 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*, MICRO '21, page 779–791, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385572.

doi: 10.1145/3466752.3480046. URL https://doi.org/10.1145/3466752.3480046.

[63] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. The agree predictor: a mechanism for reducing negative branch history interference. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, page 284–291, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919017. doi: 10.1145/264107.264210. URL https://doi.org/10.1145/264107.264210.

[64] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen. Half and half: Demystifying intel's directional branch predictors for fast, secure partitioned execution. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, 2023. doi: 10.1109/SP46215.2023.10179415.

[65] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, page 51–61, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914600. doi: 10.1145/123465.123475. URL https://doi.org/10.1145/123465.123475.

[66] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. Branchnet: A convolutional neural network to predict hard-to-predict branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 118–130, 2020. doi: 10.1109/MICRO50266.2020.00022.

[67] Siavash Zangeneh Kamali. *Using Convolutional Neural Networks to Improve Branch Prediction.* Ph.d. dissertation, The University of Texas at Austin, Austin, TX, USA, 2022. URL https://hps.ece.utexas.edu/pub/TR-HPS-2022-002.pdf.

[68] Anastasios Zouzias, Kleovoulos Kalaitzidis, Konstantin Berestizshevsky, Renzo Andri, Leeor Peled, and Zhe Wang. Identifying and exploiting sparse branch correlations for optimizing branch prediction. *CoRR*, abs/2207.14033, 2022. doi: 10.48550/ARXIV.2207.14033. URL `https://arxiv.org/abs/2207.14033`.