Enabling Ahead Prediction with Practical Energy Constraints

Lingzhe(Chester) Cai
UT Austin
Austin, TX, USA
chestercai@utexas.edu

Aniket Deshmukh
UT Austin
Austin, TX, USA
a.deshmukh@utexas.edu

Yale Patt
UT Austin
Austin, TX, USA
patt@ece.utexas.edu

Abstract

Accurate branch predictors require multiple cycles to produce a prediction, and that latency hurts processor performance. "Ahead prediction" solves the performance problem by starting the prediction early. Unfortunately, this means making the prediction without the directions of the N branches between when the prediction starts and when the relevant branch's prediction is needed. The energy required to consider all 2^n possible cases increases by 14.6x, making ahead prediction not viable. This paper shows that most of the intermediate branch directions never materialize, reducing the number of observed missing history patterns significantly (usually, only one or two). We modified the TAGE predictor to eliminate those branches from having to be considered. The result, our ahead predictor can produce a performance benefit of 4.4%, while causing an increase in energy consumption of only 1.5x, far less than the 14.6x that was thought to be necessary, and very much viable.

CCS Concepts

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

Keywords

CPU architecture, Branch Prediction

ACM Reference Format:
Lingzhe(Chester) Cai, Aniket Deshmukh, and Yale Patt. 2025. Enabling Ahead Prediction with Practical Energy Constraints. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25), June 21–25, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3695053.3730998

1 Introduction

Branch prediction remains an important bottleneck for single-thread performance. Highly accurate predictors ensure correct path instructions are sent to the processor backend. Decades of research have focused on predictor accuracy, resulting in increasingly larger storage overheads and more complex logic. This improves accuracy but leads to multi-cycle predictor lookup latency, significantly decreasing the overall throughput of the predictor.

To reduce the impact of this latency problem while maintaining high accuracy, industry uses multi-level branch prediction [2, 13, 17, 33]. 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.

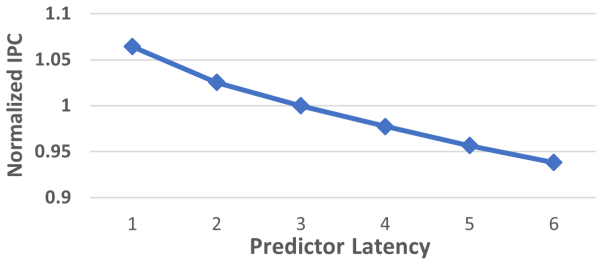

Figure 1: Performance Impact of BP Latency

Fig. 1 shows the impact of overriding predictor latency on processor performance in SPEC CPU2017 [4] benchmarks. We use a 1K-entry 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 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%.

Ahead prediction [14, 16, 26, 28, 29, 40, 41] has long been proposed as a solution to the predictor latency problem. Instead of using the history and PC at the current branch to predict the current branch, ahead prediction uses the history and PC available now to skip ahead and predict a future branch. This allows the prediction to be initiated earlier which hides the multi-cycle predictor latency. Skipping history can reduce prediction accuracy, and thus prior work [19, 37, 38] preemptively generates multiple predictions by reading out consecutive entries from the prediction table(s). Each prediction corresponds to one of the possible missing history patterns, and the final prediction is picked when the missing history is available. This increases the energy required per prediction significantly because exponentially more bits need to be read out of the prediction tables as the ahead distance increases. The design suggested in [38] increases the number of bits read out per prediction by 32x when predicting 5 branches ahead, which is necessary to hide the latency of 3 cycles. This increases per-prediction energy by 14.6x, making it infeasible to build as the directional branch predictor uses around 3-4% of the total core power [5, 6, 32].

ISCA '25, Tokyo, Japan

ACM ISBN 979-8-4007-1261-6/25/06
https://doi.org/10.1145/3695053.3730998

Our work proposes a feasible ahead predictor design that does not incur a high energy cost. Specifically:

- We show experimentally and analytically that the number of missing history patterns observed during program runtime is far less than the theoretical maximum.
- Using this insight, we propose an efficient ahead predictor design that explicitly tags each counter with its corresponding missing history.
- The per-prediction energy of our predictor scales linearly with ahead distance. We can hide the full predictor latency with only a 1.5x increase in per-prediction energy compared to prior designs that scale exponentially and require 14.6x more energy per-prediction.

The rest of the paper is organized as follows: Sec. 2 provides the necessary background. Sec. 3 provides an analysis of the number of missing history patterns. Sec. 4 provides an overview of our prediction scheme. Sec. 5 covers other implementation details. Sec. 6 provides a detailed evaluation of our design.

2 Background

2.1 Branch Prediction

Branch prediction algorithms exploit the correlation between the control flow leading up to a branch and its direction [7, 12]. These correlations can come from program control/data flow or the inherent correlation in program data. The two-level predictor [45] and its variants[27] assign a saturating counter for each control flow leading up to a branch. This captures the majority direction of this branch under the corresponding control flow. TAGE [42] improves upon this by trying to identify the shortest history length required to capture the majority direction of the branch and explicitly tagging each counter with its corresponding control flow. A branch is unpredictable if it does not exhibit stable behavior (i.e., does not have a majority direction) under the longest history supported by the predictor. For the rest of the paper, we will refer to predictable branches as branches that exhibit stable behavior under a specific control flow and unpredictable branches as branches that do not exhibit stable behavior under a specific control flow. Note that a branch can be predictable under some control flows but unpredictable under other control flows.

2.2 Branch Predictor Latency

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. For the current state-of-the-art predictor, TAGE-SC-L [39], a 64KB version achieves 25.3% fewer mispredictions compared to a 8 KB TAGE-SC-L. In terms of logic complexity, the initial two-level [45] predictor design 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 [18, 21] require table look-ups and a dot product computation for the final prediction. A recent proposal [46] 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.

2.3 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, 13, 17, 33] 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. The single-cycle predictors are usually extremely simple due to timing constraints and are, therefore, significantly less accurate than the overriding predictor. Our 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 predictor schemes also limit scaling in two ways. **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%[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 where the predictor is stalled. This decreases the effectiveness of a wider frontend, which is critical for high performance.

2.4 Baseline

Our baseline features a two-level predictor. The single-cycle predictor in our baseline is coupled with a single-cycle BTB. The two are implemented in one table containing 1K entries (4-way set associative cache). An entry for a branch in this table contains its target and a two-bit counter used for predicting its direction. We model the throughput of an aggressive frontend that can predict up to the first taken branch or 16 instructions per cycle. There are no limitations on the number of not-taken branches predicted per cycle. The fetch unit can fetch up to 16 instructions per cycle from the I-cache and does not break on cache-line boundaries. The baseline has a decoupled frontend that can buffer up to 8 prediction packets. The parameters for our baseline are summarized in Section 6.1

[1]Assuming doubling the predictor size increases predictor latency by 1

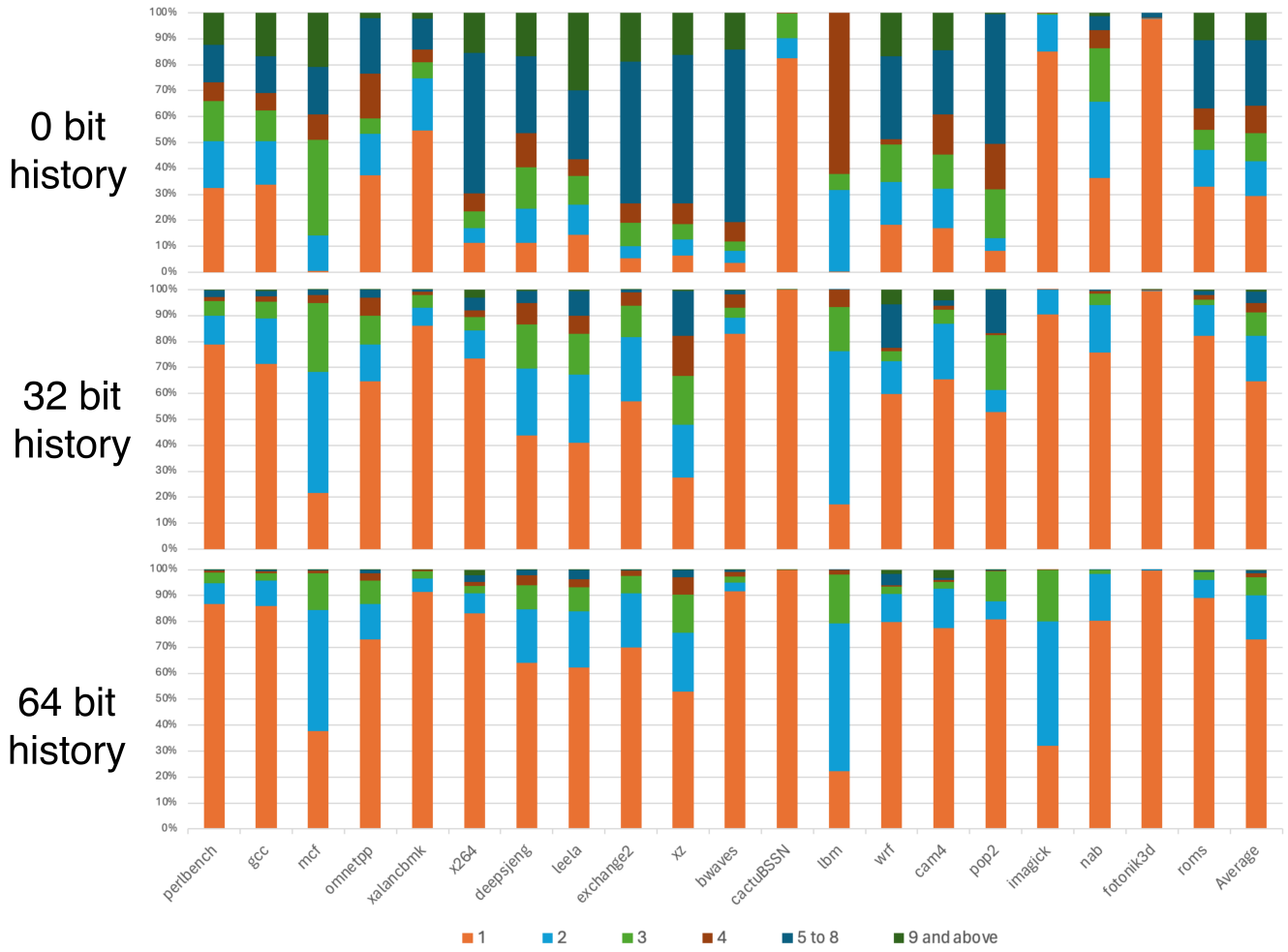

Figure 2: Number of Patterns Under Control Flow

2.5 Decoupled Frontend

While the decoupled frontend[35] 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. Our 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 3 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.

2.6 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,[2] making it hard to know which one the prediction is for.

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 [38] would need to read out 10,048 (32x314) bits per prediction (for an ahead distance of 5). Our 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.

3 Number of Missing History Patterns

While prior work starts with the assumption that every missing history pattern needs to be considered, we 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, we can leverage this fact to implement an efficient ahead predictor.

3.1 Experimental Results

We examine the benchmarks in the SPEC CPU2017 suite to answer the above question. For each branch on the correct path, we collect the control flow leading up to the branch and PC, along with the PC of the next 5 branches[3]. The control flow is captured via a global history register similar to the one used in the 2-level branch predictor[45]. The history and PC pair represent the ahead history and PC in our experiment. The sequence of the next 5 branches represents the missing history patterns observed. For each distinct control flow, we count the number of unique patterns for the next 5 branches. We use no history, 32 bits of history, and 64 bits of history, as shown in Fig. 2.

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

[3]we chose 5 branches for this experiment as this is the ahead distance used in our design

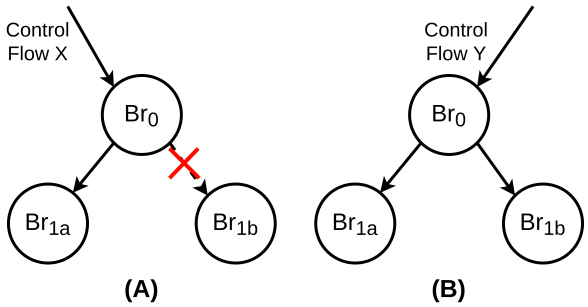

Figure 3: Control Flow Example A and B

There are 32 (2^5) possible paths for the next 5 branches, assuming they are direct conditional branches. However, significantly fewer patterns are observed in our experiment. We 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, we 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 is because the predictability of intermediate branches determines how many patterns are seen, as discussed in the following section.

3.2 Predictable Intermediate Branches

Consider the example with 3 branches, Br_0, Br_{1a}, and Br_{1b} pictured in Fig. 3-A. Br_0 is predictable under control flow X. Thus, every time the program reaches Br_0 under control flow X, Br_0 always leads to Br_{1a}. Only one path is possible going forward from control flow X.

Now consider the same example, but with multiple branches, Br_0 to Br_N. If all the intermediate branches are easy to predict under control flow X, for each Br_i where $0 \leq i < N$, the next branch will always be $Br_i + 1$. Thus, after each branch, there is only one path going forward. By repeating this process, we see that there is only 1 possible pattern after control flow X as long as all the intermediate branches are predictable with control flow X. With longer history lengths, more branches become predictable. This increases the number of cases where only very few patterns are observed when a longer history is used. Thus, we see only one pattern for more than 70% of the control flows when 64 bits of history are used, as shown in Fig. 2.

However, if there are unpredictable branches among the intermediate branches $Br_i (0 \leq i < N)$, the control flow does not always

reach Br_N. Each additional unpredictable branch increases the number of possible patterns. This is why benchmarks with high branch MPKI have more patterns, as seen in Fig. 2.

3.3 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 we use ahead history or current history. This can be achieved by training the counter attached to the ahead history with the direction of the branch we wish to predict. However, if an unpredictable branch is skipped, prediction accuracy drops significantly.

Assume we arrive 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. 3-B. If ahead prediction with 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, ahead prediction 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 we will refer to as missing history.

In the benchmarks we evaluated, most branches tend to be predictable. As seen in Fig. 2, 71% of control flows only lead to 1 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%). We use this insight to drive our ahead predictor design.

4 Designing an Efficient Ahead Predictor

Similar to prior work [19, 38], we 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, we take advantage of the fact that only a few paths show up at runtime. We take note of 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. We use these 2 characteristics by distributing entries corresponding to different missing history patterns across different TAGE tables. We identify the missing history pattern that a counter belongs to with an additional tag field, the secondary tag. Fig. 4 shows the layout of a TAGE entry in our new design. Note that T0 (the bimodal table inside TAGE) remains untagged.

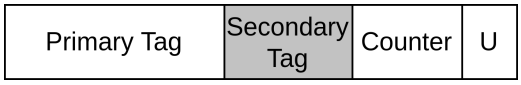

Figure 4: TAGE Entry with Secondary Tag

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 our 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 history. A mismatch in the main tag means the entry corresponds to a different ahead history and should be ignored. Our ahead predictor reads out one counter per table, similar to baseline TAGE. We 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 our predictor can distinguish and is independent of the ahead distance.

Even though we 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[38] and does not increase exponentially with the ahead distance.

Our ahead predictor design is shown in Fig. 5. It includes 26 history lengths (T0 through T25) similar to baseline TAGE. We use a 5-bit secondary tag as our 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.

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. 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 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, we compute a hash based on the direction and target of the intermediate branches. This is the secondary tag for the branch we are predicting and determines the final prediction.

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

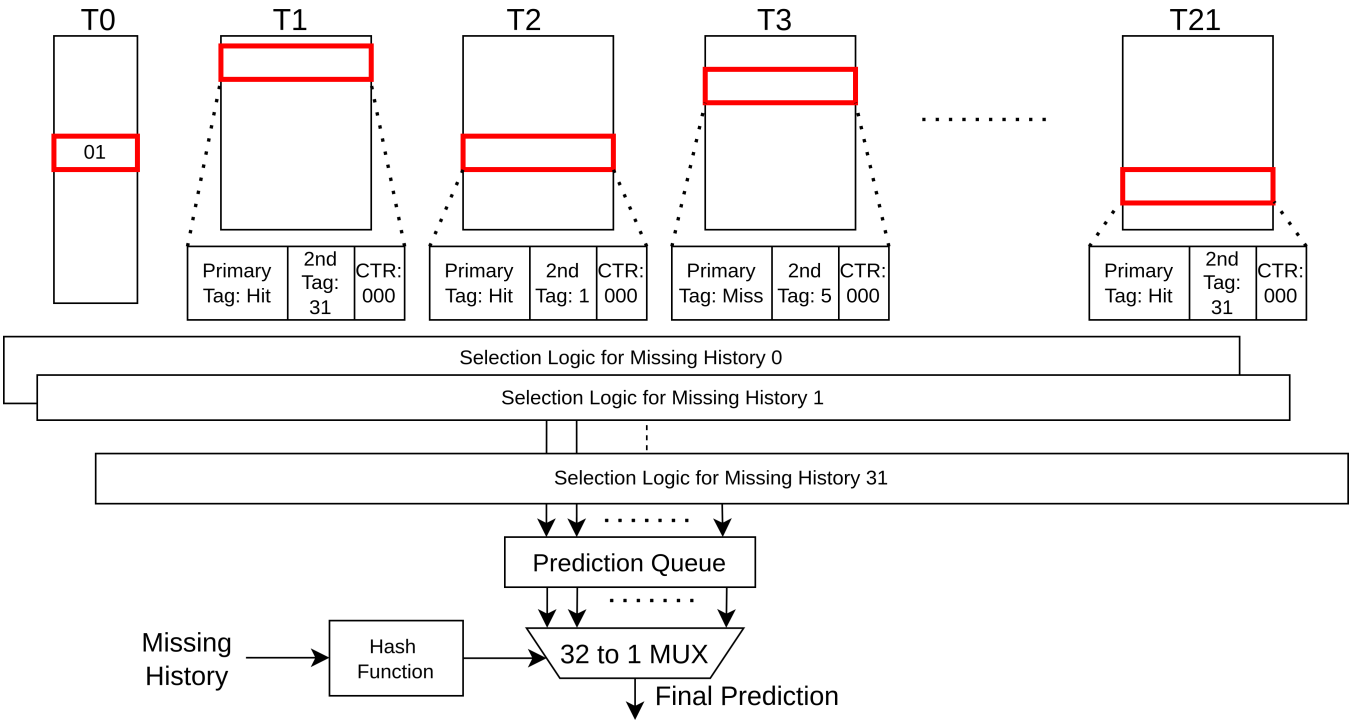

Figure 5: Ahead 2-Tag TAGE Prediction Example

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. We 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 6.3). 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. 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 our hash function is described below.

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, we again follow the baseline algorithm to find an existing entry

```
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 6: Missing History Hash Algorithm

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.

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 our 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 and hurts accuracy. However, because there are only a few (<3) patterns most of the time (97%), these conflicts are minimized and do not significantly impact prediction accuracy.

We compare the prediction accuracy of branches in our ahead predictor with baseline TAGE. We categorize each branch based on the number of missing history patterns its corresponding ahead history observes, and the results are shown in Table 1. This shows that branches with only a few missing history patterns (1-3) see very little degradation in accuracy with our 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, our ahead predictor only decreases the prediction accuracy by 0.067% compared to the baseline TAGE.

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 1: Accuracy Diff between Ahead Predictor and Baseline

4.5 Energy Comparison against Prior Work

The main difference between our predictor design and prior approaches is that we generate 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. Our 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 our 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 altpred table (16 entries), the energy required from this is significantly less than the table reads. Thus we 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 our design increases linearly with the ahead distance while the energy of prior work[38] increases exponentially. We use Cacti[30] 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 our ahead predictor, as ahead distance increases, the port size only increases by 1 in each table (except for the bimodal table). Fig. 7 shows the per-prediction energy (normalized to baseline TAGE) required for different ahead distances. The numbers show that the scaling is much better for our design which makes it practical to implement even for large ahead distances.

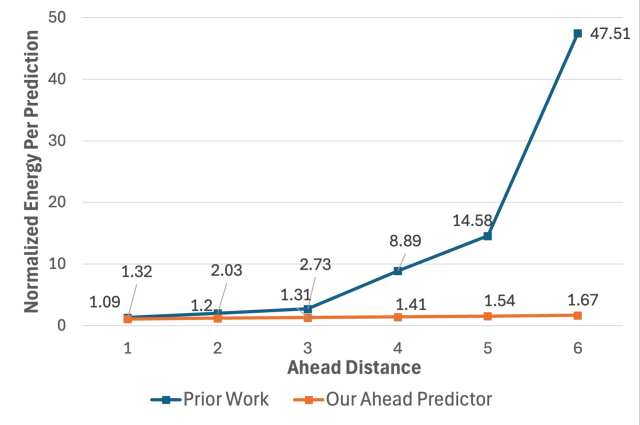

Figure 7: Normalized Energy vs. Ahead Distance

5 Integration with Core

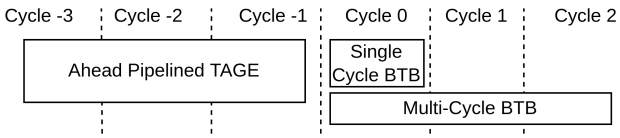

Figure 8: Prediction Timing Diagram

Fig. 8 shows the timing diagram for predicting Branch N. Cycle 0 is when the PC and full history of branch N becomes 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. 9 shows the ahead prediction pipeline. The ahead predictor uses the current PC and current history to generate a prediction for the branch exactly N ahead (branch N is predicted with the PC and control flow at branch 0). The ahead predictor generates 2^M

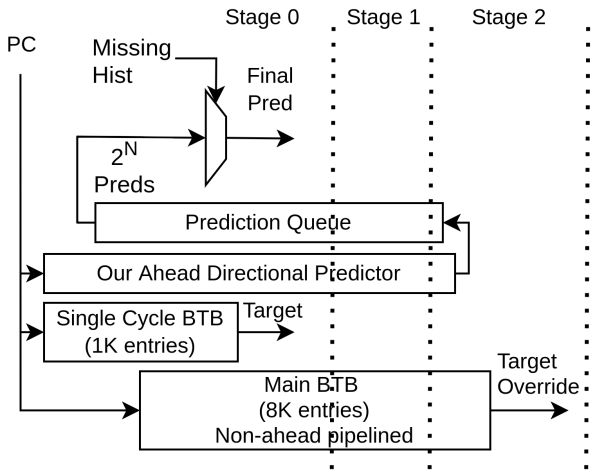

Figure 9: Prediction Pipeline

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.

5.1 Single-Cycle Override

Both prior work[38] and our 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. We 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. We 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 SPEC benchmarks.

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 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 pointer 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.

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, we compare it against the single-cycle prediction. If this prediction agrees with the bi-modal 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 provides 2.4% of IPC improvement.

5.4 Prediction Pipeline Restart

Our design handles misprediction flushes by manipulating the read, write, and allocation pointers to the prediction queue. For each branch, we 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 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 [38], our 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, we achieve high prediction accuracy right after a pipeline flush at minimum overhead.

Fig. 10 shows what happens if Br_X is mispredicted by our ahead predictor. We 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.

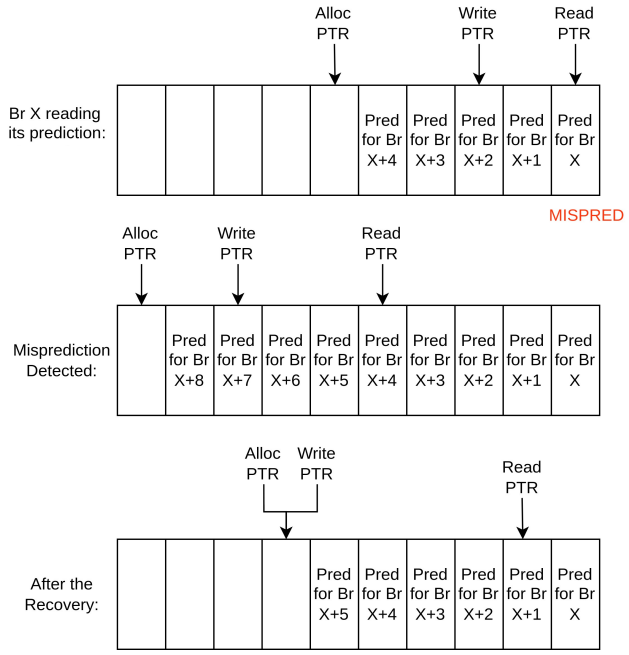

Figure 10: Prediction Queue Recovery Example

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, our final selection logic does not depend on the predictions itself, 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 queue and the propagation delay through the muxes, and should not increase the critical path of that stage.

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, our design adds 19.65KB of area overhead. Note that we provide the comparison between our design and a large baseline TAGE to match the storage in Sec. 6.6.

6 Evaluation

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 Capcity: 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: Simulation Parameters

To evaluate how our predictor design affects prediction accuracy and overall processor performance, we simulate the microarchitecture 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 our ahead predictor are listed in Table 2. Our baseline models a very aggressive OoO core that contains a multi-level predictor (Section 2.2) and a decoupled frontend.

We 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 [39]. 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.

We use all applications from SPEC CPU2017 (speed) in our evaluation. We use SimPoints [43] to generate up to 5 Simpoints for each input set, with 200 million instructions per Simpoint.

6.2 Results

We first evaluate the performance impact of our ahead predictor compared to the baseline TAGE. Fig. 11 shows the MPKI of baseline

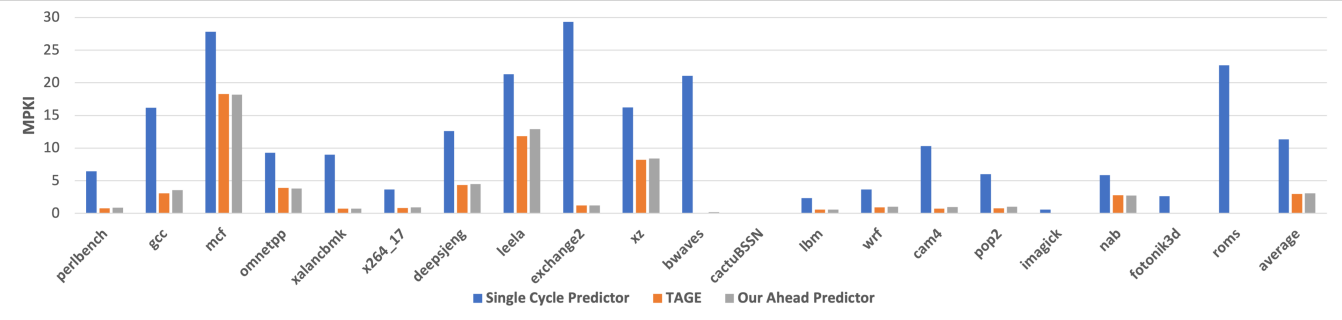

Figure 11: MPKI of Baseline TAGE and Our Ahead Predictor

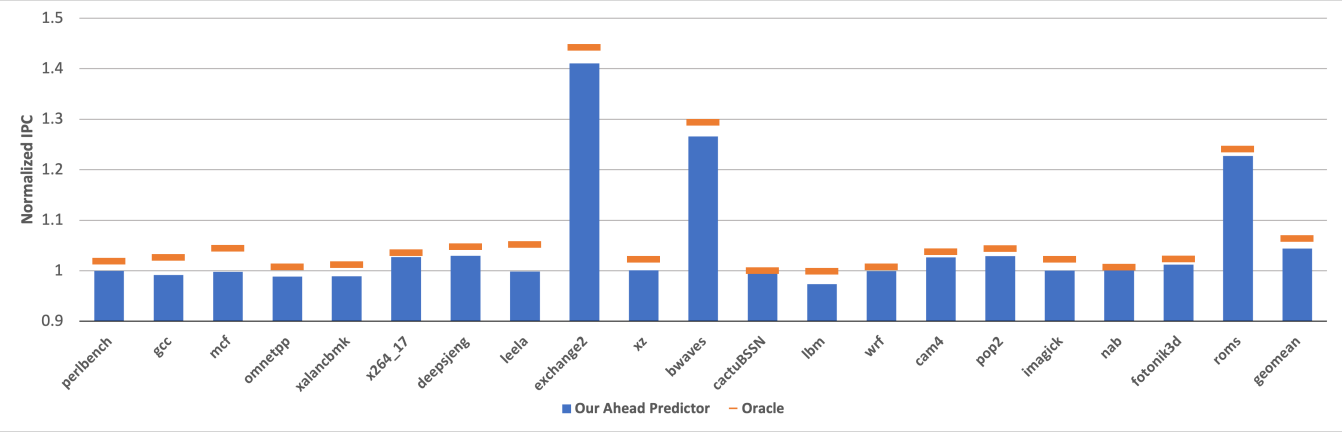

Figure 12: Normalized Performance Improvement

single cycle predictor, baseline TAGE and our ahead predictor. On average, our predictor is within 0.1 MPKI of the baseline TAGE. Fig. 12 shows the normalized performance of using our ahead predictor compared to a baseline out-of-order core with multi-level prediction. We also compare our scheme against an oracle where the TAGE latency is reduced to 1 cycle and is used as the single cycle predictor. Overall, our 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 back-end 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 our design is unable to capture all the missing history patterns as there are many clustered unpredictable branches in these benchmarks. In omnetpp and xalancbmk, our ahead predictor has a better MKPI 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 our ahead predictor as explained in Sec. 5.1, decreases the predictor accuracy, and adversely affects performance.

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

Sensitivity to fetch queue size: The performance does not decrease much with larger queue sizes. Increasing the number of entries from 8 to 20 only dropped performance to 4%.

6.3 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. We evaluate our design at an ahead distance of 5 with 0 to 9 bits of secondary tag. A tag width of 0 means that the predictor just uses ahead information and does not use any missing history information.

Fig. 13 shows the MPKI of our 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

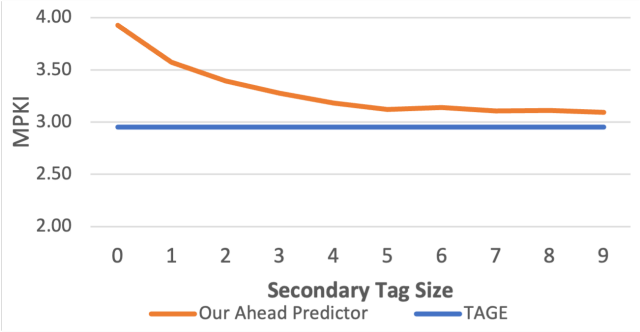

Figure 13: MPKI vs Secondary Tag Size

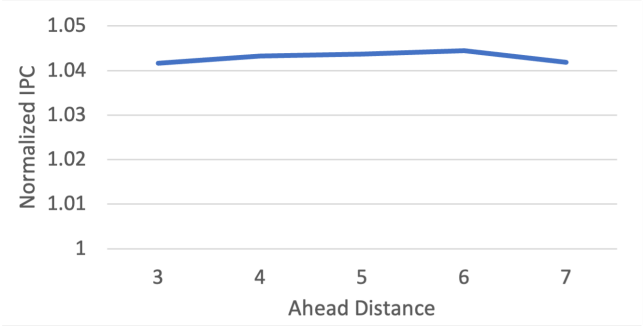

Figure 15: Normalized IPC vs Ahead Distance

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. We 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.

6.4 Ahead Distance

Fig. 14 shows the MPKI of our ahead predictor and prior work as ahead distances increases from 3 to 7. With a longer ahead distance, more missing history patterns are exposed. Our 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. 4.5, makes prior work impossible to be implemented.

Our 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. 15 shows the normalized IPC of our predictor as the ahead distance increases (the secondary tag increases with the ahead distance). In our design, an ahead distance of 6 covers almost all of the predictor latency.

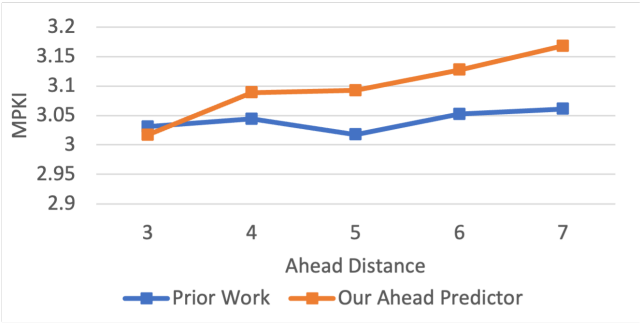

Figure 14: MPKI vs Ahead Distance

6.5 MPKI vs Number of Tables Read

Fig. 16 shows the MPKI of our ahead predictor as the number of tables read per prediction decreases from 21 to 14. We 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, we 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.

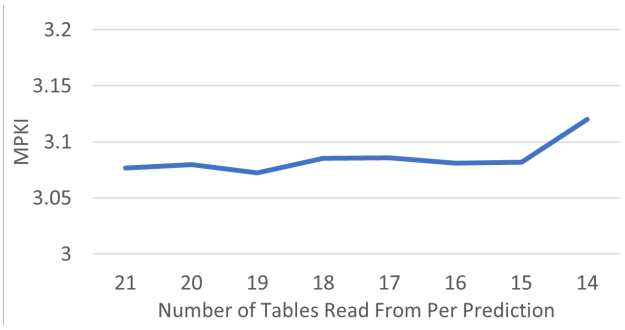

Figure 16: MPKI vs Number of Histories

6.6 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 our ahead prediction scheme offers.

7 Related Work

Unlike the TAGE predictor that attaches a prediction counter to a particular control flow, Perceptron Branch Predictor [18] uses a weight for each bit in the history representing its contribution to the overall prediction and sums all the weight up at the time of prediction. The latency of Perceptron is later improved [15, 20] 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.

Recent work [10] 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 [22, 23, 25, 34, 44], very few works have been published on predictor capacity. Whisper [24] 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[36] proposes using secondary storage to back up the main predictor, and a prefetcher to manage the secondary storage. While our 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 6.4. In addition, the physical location of the main predictor is near the processor frontend, where the on-chip area is extremely contentious. A prefetcher approach like the Last-Level Branch Predictor [36] can enable larger predictor designs beyond what an ahead prediction approach can provide. We believe all three approaches can be combined in the future.

Our 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 [35] 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 [9], CDF [11], Precise Runahead Execution [31], and TEA [8] 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.

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.

Our paper shows that only a few missing history patterns are observed in the program's runtime. Using this insight, we present a new approach for building ahead predictors that does not require reading exponentially more entries for large ahead distances. Our 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, our work allows for larger and more complex predictors and better pipelining width scaling.

Acknowledgments

We thank the anonymous reviewers, the members of the HPS Research Group for their feedback and help in improving this paper. We also thank Intel, Arm, and Rivos for their financial support.

References

[1] [n. d.]. Championship Branch Prediction. https://jilp.org/cbp2016/.
[2] [n. d.]. Loongson 3A6000: A Star among Chinese CPUs. https://chipsandcheese.com/2024/03/13/loongson-3a6000-a-star-among-chinese-cpus/.
[3] [n. d.]. Scarab. https://github.com/hpsresearchgroup/scarab.
[4] 2017. The Standard Performance Evaluation Corporation (SPEC). https://www.spec.org/cpu2017/
[5] Daniel Chaver, Luis Piñuel, Manuel Prieto, Francisco Tirado, and Michael C. Huang. 2003. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design* (Seoul, Korea) *(ISLPED '03)*. Association for Computing Machinery, New York, NY, USA, 390–395. doi:10.1145/871506.871603
[6] Daniel Chaver, Luis Piñuel, Manuel Prieto, Francisco Tirado, and Michael C. Huang. 2003. Branch prediction on demand: an energy-efficient solution. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design* (Seoul, Korea) *(ISLPED '03)*. Association for Computing Machinery, New York, NY, USA, 390–395. doi:10.1145/871506.871603
[7] Jian Chen and Lizy K. John. 2011. Autocorrelation analysis: A new and improved method for branch predictability characterization. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 194–203. doi:10.1109/IISWC.2011.6114179
[8] Aniket Deshmukh, LingzheChester Cai, and Yale N. Patt. 2024. Timely, Efficient, and Accurate Branch Precomputation. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 480–492. doi:10.1109/MICRO61859.2024.00043
[9] Aniket Deshmukh, Lingzhe Chester Cai, and Yale N. Patt. 2024. Alternate Path Fetch. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1217–1229. doi:10.1109/ISCA59077.2024.00091
[10] Aniket Deshmukh, Ruihao Li, Rathijit Sen, Robert R. Henry, Monica Beckwith, and Gagan Gupta. 2021. Performance Characterization of .NET Benchmarks. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 107–117. doi:10.1109/ISPASS51385.2021.00028
[11] Aniket Deshmukh and Yale N. Patt. 2021. Criticality Driven Fetch. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 380–391. doi:10.1145/3466752.3480115
[12] M. Evers, S.J. Patel, R.S. Chappell, and Y.N. Patt. 1998. 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)*. 52–61. doi:10.1109/ISCA.1998.694762
[13] Brian Grayson, Jeff Rupley, Gerald Zuraski Zuraski, Eric Quinnell, Daniel A. Jiménez, Tarun Nakra, Paul Kitchin, Ryan Hensley, Edward Brekelbaum, Vikas Sinha, and Ankit Ghiya. 2020. Evolution of the Samsung Exynos CPU Microarchitecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 40–51. doi:10.1109/ISCA45697.2020.00015
[14] Yasuo Ishii. 2007. 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 (2007), 1–19.
[15] D.A. Jimenez. 2003. Fast path-based neural branch prediction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 243–252. doi:10.1109/MICRO.2003.1253199
[16] D.A. Jimenez. 2005. Piecewise linear branch prediction. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 382–393. doi:10.1109/ISCA.2005.40
[17] D.A. Jimenez, S.W. Keckler, and C. Lin. 2000. The impact of delay on the design of branch predictors. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000.* 67–76. doi:10.1109/MICRO.2000.898059
[18] D.A. Jimenez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture.* 197–206. doi:10.1109/HPCA.2001.903263
[19] Daniel A. Jiménez. 2003. Reconsidering Complex Branch Predictors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*. IEEE Computer Society, USA, 43.
[20] Daniel A. Jiménez. 2005. Improved latency and accuracy for neural branch prediction. *ACM Trans. Comput. Syst.* 23, 2 (may 2005), 197–218. doi:10.1145/1062247.1062250
[21] Daniel A. Jim´enez. 2016. Multiperspective Perceptron Predictor with TAGE. https://api.semanticscholar.org/CorpusID:221213615
[22] Cansu Kaynak, Boris Grot, and Babak Falsafi. 2015. Confluence: Unified instruction supply for scale-out servers. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 166–177. doi:10.1145/2830772.2830785
[23] Tanvir Ahmed Khan, Nathan Brown, Akshitha Sriraman, Niranjan K Soundararajan, Rakesh Kumar, Joseph Devietti, Sreenivas Subramoney, Gilles A Pokam, Heiner Litz, and Baris Kasikci. 2021. Twig: Profile-Guided BTB Prefetching for Data Center Applications. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 816–829. doi:10.1145/3466752.3480124
[24] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A. Jiménez, and Baris Kasikci. 2022. Whisper: Profile-Guided Branch Misprediction Elimination for Data Center Applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 19–34. doi:10.

1109/MICRO56248.2022.00017

[25] Rakesh Kumar, Boris Grot, and Vijay Nagarajan. 2018. Blasting through the Front-End Bottleneck with Shotgun. *SIGPLAN Not.* 53, 2 (mar 2018), 30–42. doi:10.1145/3296957.3173178

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

[27] Scott Mcfarling. 1998. Combining Branch Predictors. (10 1998).

[28] P. Michaud, A. Seznec, and S. Jourdan. 1999. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00425)*. 2–10. doi:10.1109/PACT.1999.807388

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

[30] Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. 3–14. doi:10.1109/MICRO.2007.33

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

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

[33] 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. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020), 53–62. doi:10.1109/MM.2020.2972222

[34] Arthur Perais and Rami Sheikh. 2023. Branch Target Buffer Organizations. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 240–253. doi:10.1145/3613424.3623774

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

[36] David Schall, Andreas Sandberg, and Boris Grot. 2024. The Last-Level Branch Predictor. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 464–479. doi:10.1109/MICRO61859.2024.00042

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

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

[39] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*. Seoul, South Korea. https://inria.hal.science/hal-01354253

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

[41] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. 1996. Multiple-Block Ahead Branch Predictors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. Association for Computing Machinery, New York, NY, USA, 116–127. doi:10.1145/237090.237169

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

[43] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior *(ASPLOS X)*. 45–57. doi:10.1145/605397.605403

[44] Niranjan K Soundararajan, Peter Braun, Tanvir Ahmed Khan, Baris Kasikci, Heiner Litz, and Sreenivas Subramoney. 2021. PDede: Partitioned, Deduplicated, Delta Branch Target Buffer. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 779–791. doi:10.1145/3466752.3480046

[45] Tse-Yu Yeh and Yale N. Patt. 1991. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture* (Albuquerque, New Mexico, Puerto Rico) *(MICRO 24)*. Association for Computing Machinery, New York, NY, USA, 51–61. doi:10.1145/123465.123475

[46] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. 2020. BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 118–130. doi:10.1109/MICRO50266.2020.00022