RUNLTS: Register-value-aware Predictor Utilizing Nested Large Tables

Toru Koizumi
koizumi@nitech.ac.jp
Nagoya Institute of Technology
Aichi, Japan

Toshiki Maekawa
maekawa@matlab.nitech.ac.jp
Nagoya Institute of Technology
Aichi, Japan

Masanari Mizuno
m_mizuno@matlab.nitech.ac.jp
Nagoya Institute of Technology
Aichi, Japan

Maru Kuroki
kuroki@rsg.ci.i.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan

Tomoaki Tsumura
tsumura@acm.org
Nagoya Institute of Technology
Aichi, Japan

Ryota Shioya
shioya@ci.i.u-tokyo.ac.jp
The University of Tokyo
Tokyo, Japan

Abstract

In this paper, we propose register-value-aware predictor utilizing nested large tables (RUNLTS). RUNLTS is a branch predictor based on TAGE-SC-L. We introduce a novel component that exploits correlations between branch directions and register values. We also introduce novel methods for determining history-length sets and controlling entry allocation, enabling the predictor to leverage larger hardware resources. We evaluate RUNLTS using the latter halves of the 105 training traces provided by CBP 2025. The evaluation results show that, on average, RUNLTS achieves a BrMisPKI of 3.197 and a CycWpPKI of 140.3.

1 Introduction

The number of in-flight instructions in recent high-performance CPUs has increased significantly, making branch prediction more critical than ever. At CBP5 in 2016, the most powerful CPUs featured a fetch width of four instructions and a reorder buffer (ROB) capable of storing approximately 200 instructions. In contrast, today's most powerful CPUs have a fetch width of ten instructions and an ROB capable of storing over 500 instructions. In such large-scale processors, the number of instructions flushed due to branch mispredictions often exceeds one hundred instructions, and the impact of a single branch misprediction on performance and power consumption is greater than ever.

In this context, we propose RUNLTS, a branch predictor based on TAGE-SC-L [9], which is widely regarded as the best-performing predictor to date, and we introduce the following techniques to achieve further performance improvements:

- We propose a novel prediction method that exploits correlations between register values and branch directions (Section 4.2). Our method generates a short digest characterizing each register value and improves prediction accuracy by capturing correlations between these digests and branch directions. We implemented this method by integrating it into the statistical corrector (SC) of the existing TAGE-SC-L predictor.

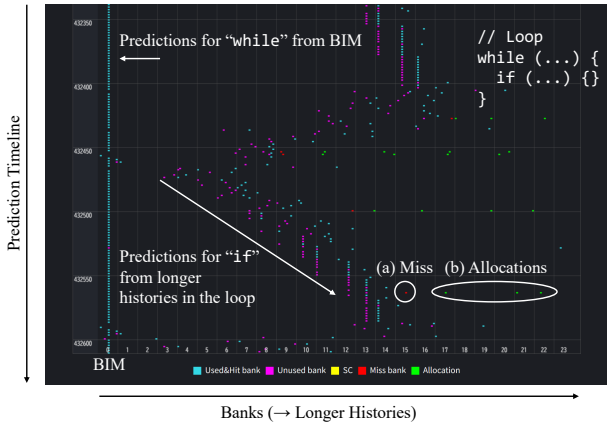

Figure 1: TAGE bank viewer (web_7 trace).

- To scale the predictor to larger configurations, we introduced a large bimodal table (Section 3.1.1), a novel history-length set that differs from the conventional geometric history-length scheme (Section 4), and a new technique for controlling entry allocation (Section 4.1).
- We incorporate the improved variant of the innermost loop iteration (IMLI) recently proposed in [11]. In particular, we found that assigning a greater weight to IMLI predictions significantly improves overall prediction accuracy, and we have integrated this enhancement into our predictor.
- We revisit the call-stack-based history mechanism [8, 15] and the pipeline-aware-predictor updates of FTL++ [2], incorporating both into our predictor.

2 Background

In this section, we present various analyses of existing predictors and the provided traces, along with the insights derived from them.

2.1 Preliminary Analysis

We developed several supporting tools to facilitate the analysis of candidate predictors and traces. For example, we implemented a

'CBP 2025', June 21, 2025, Tokyo, Japan

tool to visualize predictor table accesses (Figure 1), developed a log generator for the pipeline visualization tool [14], and created a utility to reconstruct assembly code from traces. Using these tools, we analyzed the traces to identify the programs that generated them. Specifically, we found that several traces originate from benchmark suites such as Speedometer, SunSpider, SPECjbb, and SPEC CPU. These analyses provided valuable insights for understanding predictor behavior and improving prediction accuracy.

2.2 TAGE and BATAGE

We considered two baseline predictors: TAGE from CBP5 [9] and BATAGE [4].

(1) TAGE is widely regarded as the most accurate branch predictor, primarily comprising multiple tagged tables. TAGE-SC-L enhances TAGE by adding a statistical corrector (SC), a variant of the hashed perceptron predictor, as well as a loop predictor, in order to further improve prediction accuracy.
(2) BATAGE is a predictor with a simpler design, achieving prediction accuracy comparable to TAGE. BATAGE uses the same tag-based table structure as TAGE. While TAGE relies on a single saturating counter to make predictions, BATAGE employs two independent counters, one for taken outcomes and one for not-taken outcomes, to estimate prediction confidence.

We initially selected BATAGE as a baseline candidate because its tables each output a confidence metric, making it easy to integrate into various hybrid predictors. We also chose BATAGE for its simplicity: it always uses the tag-match output with the highest confidence, whereas TAGE dynamically selects between two tag-match outputs.

However, we found that BATAGE does not scale as well as TAGE. To improve accuracy under the large hardware budgets available in CBP 2025, it is essential to allocate multiple entries upon each misprediction. We discovered that the useful bit (u-bit) of TAGE is crucial in large-scale predictors that employ such aggressive learning. The u-bit is set not according to whether its prediction was correct but according to whether the entry is deemed *useful*; it is set if omitting that entry would have caused a misprediction. Only useful entries are protected from replacement, while others remain subject to eviction. This mechanism ensures that only the most necessary entries are retained, while redundant entries are evicted.

In contrast, BATAGE determines whether an entry can be overwritten based on the prediction confidence of each entry. The confidence increases when the prediction is correct; thus, the confidence does not necessarily reflect the actual usefulness of the entry. We observed that, when the predictor is scaled up, BATAGE tends to retain unnecessary entries and does not scale as efficiently as TAGE.

3 High-level Design Overview

Based on these observations, we selected TAGE-SC[1] from CBP5 as our baseline. We then applied several enhancements to further improve its accuracy. Figure 2 shows an overview of RUNLTS. The

[1]The loop predictor is omitted in our design because of its high complexity and low contribution to the prediction accuracy.

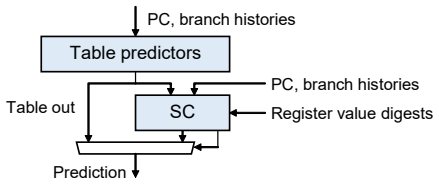

Figure 2: High-level overview of RUNLTS.

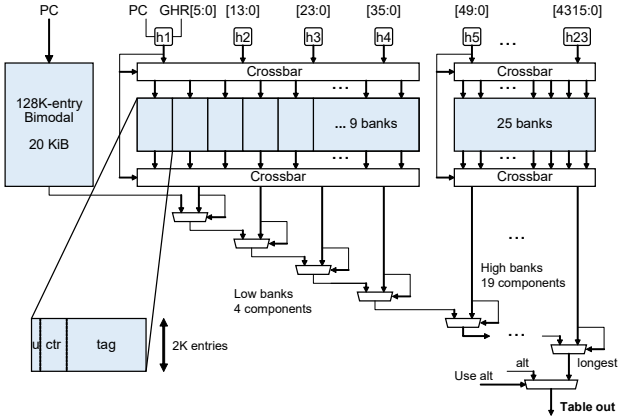

Figure 3: Structure of table predictors, consisting of a large Bimodal predictor and tagged tables with bank interleaving.

basic structure of RUNLTS is the same as that of TAGE-SC, consisting of multiple tagged tables and an SC. To exploit correlations with register values, we feed not only the PC and branch history but also *register-value digests* into the SC.

3.1 Bimodal and Tagged Tables

3.1.1 Large Base Predictor. As shown on the left-hand side of Figure 3, RUNLTS uses a Bimodal predictor with 128 K entries (20 KiB) capacity as its base predictor. This is more than a fivefold increase compared with the 8 K-entry (1.25 KiB) Bimodal predictor used in the 64 KiB TAGE-SC-L design from CBP5, although the total storage budget has increased by only a factor of three.

This large base predictor can effectively handle the ever-growing instruction footprints of modern applications. In dynamic languages such as JavaScript, instruction footprints are significantly larger than those of conventional programs. Addressing such large footprints has become a critical challenge in modern processor front-end design for both server- and client-based applications [3, 5, 7, 10]. This enlarged base predictor thus provides a simple and efficient means to handle these large footprints.

3.1.2 Novel History Lengths. We introduce a novel method for selecting history lengths tailored to large-scale predictors. Conventional predictors such as TAGE and BATAGE commonly use geometric history lengths. Additionally, CBP5 TAGE-SC-L employs an approach that sparsifies very short and very long histories while densely allocating medium histories using skewed-associative patterns, as described by Seznec [11]. In contrast, we found that the method that combines a second-order arithmetic progression with a geometric progression (Figure 4) is near-optimal, and we have

History length	0	6	14	24	36	50	66	84	104	126	150	178	212	252	300	358	426	506	602	776	1078	1606	2554	4316
First difference		+6	+8	+10	+12	+14	+16	+18	+20	+22	+24	×1.19	×1.19	×1.19	×1.19	×1.19	×1.19	×1.19	×1.19	×1.29	×1.39	×1.49	×1.59	×1.69	
Second difference		2	2	2	2	2	2	2	2	2									0.1	0.1	0.1	0.1	0.1	

Figure 4: Our novel history lengths selection, combining a second-order arithmetic progression and a geometric progression.

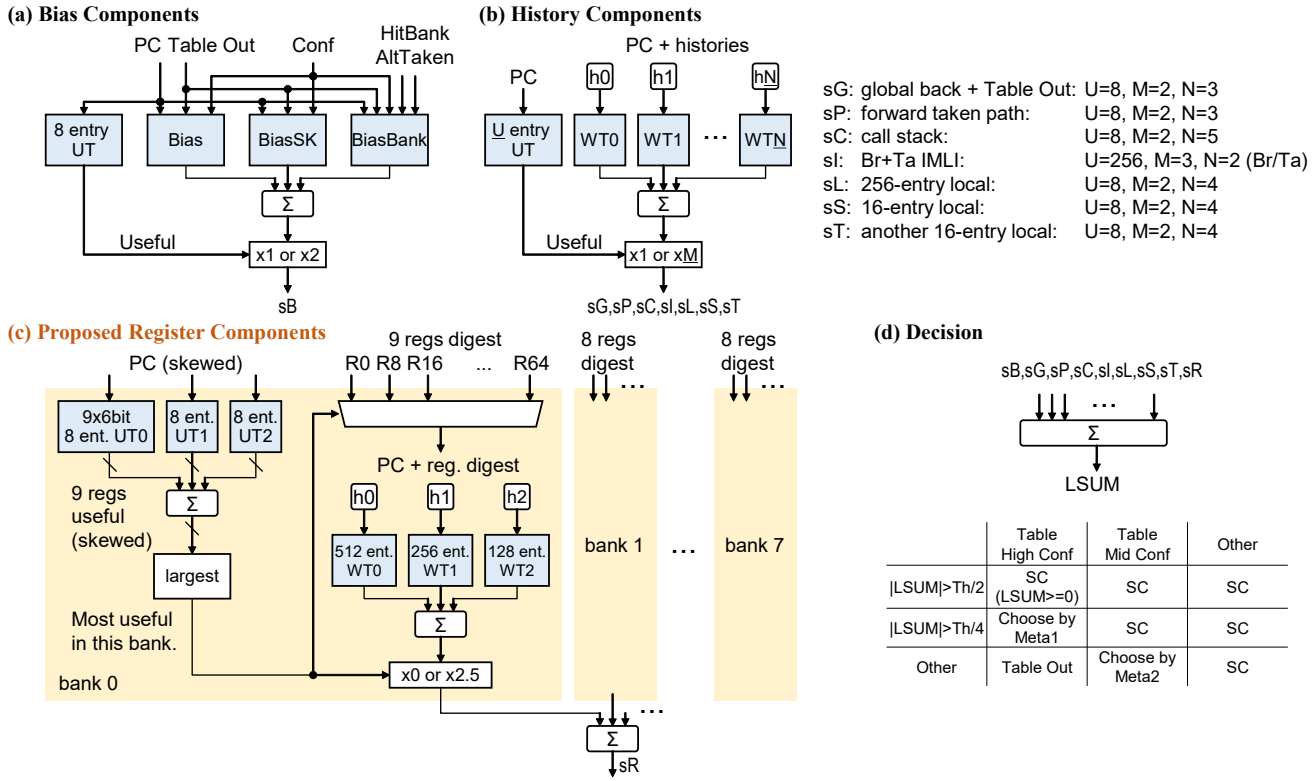

Figure 5: Design of statistical corrector (SC) predictor.

adopted it. Note that, unlike CBP5 TAGE-SC-L, our configuration does not use skewed-associative tables.

3.2 Statistical Corrector Structure

Figure 5 shows the structure of the SC in RUNLTS. Although the SC uses the same components as those in the TAGE-SC-L predictor from CBP5, we introduced the following extensions:

(1) `sC`: Call-stack-based history [8]
(2) `sI`: BrIMLI and TaIMLI [11]
(3) `sR`: Our proposed component for exploiting register-value correlations (Section 4.2)

We also discovered that assigning greater weight to useful predictions in the `sI` component significantly improves accuracy, and we incorporated this mechanism into RUNLTS.

4 Predictor Operation

The predictor's operation remains almost the same as that of TAGE-SC-L. This section summarizes the novel elements we introduced.

4.1 Tagged-Table Allocation

To make effective use of the 192 KiB capacity and to quickly respond to new program phases, we dynamically adjust the entry allocation of TAGE. In conventional TAGE, only one entry is allocated per branch misprediction [12]. While this approach minimizes the table footprint and is nearly optimal for small predictors, it does not hold for large predictors. For example, the 64 KiB TAGE-SC-L from CBP5 allocates two entries per branch misprediction [9]. A more aggressive allocation can benefit predictors with a 192 KiB capacity. However, if it is too aggressive, it may evict useful information, and thus some form of allocation throttling is desirable.

We propose a thrashing-detection method for TAGE that requires no extra storage and can adjust the allocation rate on the fly. While the BATAGE predictor can detect thrashing, detecting it in TAGE has remained an open problem [4]. We observed that combinations in which the prediction counter is 0 or −1 and the u-bit is set rarely appear. To repurpose these combinations, we force the u-bit unset whenever these combinations arise; we verified that this change has virtually no impact on prediction accuracy. Then, we can safely re-purpose these combinations as an explicit marker for newly allocated entries: when an entry is allocated, we initialize

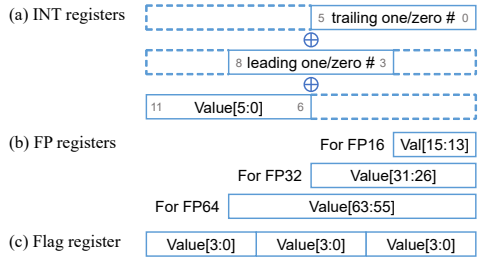

Figure 6: Method for generating a digest from register values.

its counter to 0 or −1 and set the u-bit to 1, and we clear the u-bit as soon as the entry is referenced. To detect thrashing using this marker, we add two counters that track (1) the number of times these newly allocated entries lead to correct predictions and (2) the number of times they are evicted without being referenced. Entry allocation rate is then adjusted based on the ratio of successful predictions to evictions.

4.2 Register Components

We extended the SC with components that capture correlations between register values and branch outcomes. The components use *digests* of currently available register values as their input.

While the register value components can contribute to prediction accuracy when the instructions that generate register values are sufficiently distant, they are particularly effective *immediately after a branch misprediction*. In general, it is difficult for predictors to observe the execution results of instructions within ∼100 prior instructions before the branch at prediction time, making it hard to exploit their correlation with branch direction. However, we found that by the time a branch resolves and its misprediction is revealed, many in-flight instructions have already finished execution, thereby enlarging the set of values visible to the predictor when it restarts, and this fact can be exploited. By leveraging these extra values after the flush, our scheme can correct subsequent mispredictions starting with the second one.

To keep track of which logical registers hold a valid digest at prediction time, RUNLTS adopts a method similar to Tomasulo's algorithm. We use a dedicated table that contains one entry per logical register. Each entry consists of a valid bit and a payload that can hold either a 14-bit tag or a 12-bit digest. At decode, an instruction that writes a register stores its ROB index as a tag in the corresponding entry and clears the valid bit. When the instruction completes, we overwrite the payload with the digest of the produced value and set the valid bit. Whenever the branch predictor queries the table, any entry whose valid bit is set forwards its digest to the register-value components in the SC for prediction.

Figure 6 shows the method we propose for generating digests of register values. RUNLTS generates a 12-bit digest from each 64-bit register value, with the format depending on the register type. (1) For integer registers, the digest encodes the count of leading zeros (or ones), the count of trailing zeros (or ones), and the six least significant bits, effectively capturing characteristics of addresses and round numbers. (2) For floating-point registers, we inspect the most significant bits to classify the format and then extract the sign bit along with the most significant bits of the exponent. (3) For condition-code flags, we replicate the four flag bits three times to fill the 12-bit digest.

Figure 5(c) shows the register components, which exploit correlations between register values and branch directions. This component is organized into eight banks, each corresponding to eight or nine logical registers. As with the SC's existing components, it comprises two tables: one that tracks prediction usefulness and one that determines the predicted branch direction.

Unlike the existing components, this component picks the digest of the most useful register among those available and forwards it to the second table. Each entry of the first table holds an eight- (or nine-) element vector of weights reflecting the usefulness of each register. This structure enables the selection of the most useful digest without adding extra memory ports.

During training, we update the correlations only for logical registers that were available at prediction time. If several registers were available within the same bank, it randomly selects one of them for the update. Owing to this training method, each table in the register component also needs just a single write port.

Preliminary experiments revealed that stale digests degrade prediction accuracy. Consequently, RUNLTS invalidates each digest after 256 subsequent instructions have been decoded, ensuring that only up-to-date information influences future predictions.

5 Discussion

Due to its structural similarity to TAGE-SC, RUNLTS is expected to exhibit comparable cold-start performance and training sensitivity. Given that our tables and core structures closely mirror those of TAGE-SC, we anticipate a similar level of implementation complexity. Integrating the path for sending register digests from the execution unit to the predictor and recovering the Tomasulo-like table may present some challenges; we plan to address these in future work.

6 Experimental Results and Analysis

We evaluated the proposed branch predictor and compared it against existing predictors using the CBP2025 simulator [13]. We used the default configuration, which simulates a processor with a fetch width of 16 and a frontend depth of 10. The first half of each trace was used for warm-up, and the second half for measurement.

Table 1 compares RUNLTS with representative existing predictors. The evaluation results show that, on average, RUNLTS achieves a BrMisPKI of 3.197 and a CycWpPKI of 140.3. Even RUNLTS without local history achieved a significant misprediction reduction over the existing predictors using local histories, demonstrating the usefulness of our proposed method.

Figure 7 breaks down the contribution of RUNLTS by feature, showing stacked MPKI deltas relative to our tuned 192 KiB TAGE-SC-L. For each feature, we measured the difference between a 192 KiB predictor with the feature and one without it. Because the 192 KiB budget is fixed, the extra storage consumed by a feature can occasionally make its contribution negative.

The register components yielded the largest and broadest gain, improving every trace category. Adding the IMLI components reduced MPKI by at least one in three traces and delivered notable

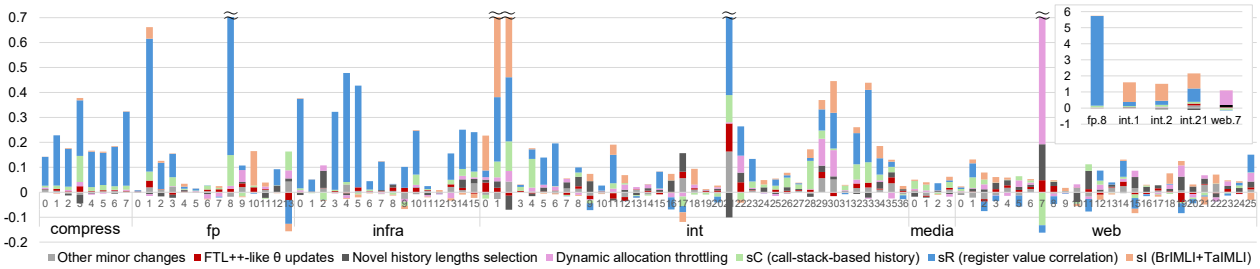

Figure 7: Feature contributions to the MPKI reduction over the TAGE-SC-L baseline. The five traces with exceptionally large improvements are plotted with a broken y-axis; the inset (upper right) shows the full-scale view. Higher is better.

Table 1: Branch predictors, features used, prediction accuracy (mispredictions per kilo instructions: MPKI), and the average penalty (wrong path cycles per kilo instructions: WPC/KI). The 'Feature' column flags features that may involve additional implementation complexity in practical implementations: 'N', 'L', and 'BI' denote neural adders, local histories, and bank interleaving, respectively. The symbol † indicates the predictors tuned by the authors.

Predictor name	Storage	Feature	MPKI	WPC/KI
GEHL perceptron†	192 KiB	N	4.088	162.9
TAGE-SC-L [9, 13]	64 KiB	BI, N, L	3.751	152.5
TAGE†	192 KiB	BI	3.674	152.1
BATAGE†	192 KiB	BI	3.667	151.9
BATAGE-GSC†	192 KiB	BI, N	3.590	149.5
TAGE-GSC†	192 KiB	BI, N	3.533	148.4
BATAGE-SC†	192 KiB	BI, N, L	3.462	146.4
TAGE-SC-L [13]	192 KiB	BI, N, L	3.428	145.4
TAGE-SC-L† (baseline)	192 KiB	BI, N, L	3.408	145.2
Prop. w/o local	192 KiB	BI, N	**3.269**	**141.5**
Proposed	192 KiB	BI, N, L	**3.197**	**140.3**

improvements in several others. Incorporating the call-stack history also helped across a wide range of traces, although part of this improvement stems from the tables acting as a conventional global GEHL perceptron in code unrelated to function calls. The dynamic allocation throttling boosts accuracy on large-footprint traces while leaving small-footprint traces essentially unchanged. Although the performance gains from the other features are not notable, each of the seven features in Figure 7 improves the 105-trace average by at least 0.005 MPKI and increases accuracy on at least four-sevenths of the traces individually.

Figure 8 shows the improvement of each trace in prediction accuracy from the TAGE-SC-L baseline for RUNLTS. RUNLTS improved prediction accuracy in all but 7 of the 105 traces. RUNLTS improved prediction accuracy by 0.052 MPKI at the median and 0.323 MPKI at the first octile.

7 Related Work

Global branch predictors, most notably TAGE [9] and BATAGE [4], offer the highest baseline accuracy, yet incorporating auxiliary information can boost performance further. Several studies leverage values produced in the execution backend, an approach orthogonal to history-based prediction. Many of these studies typically track

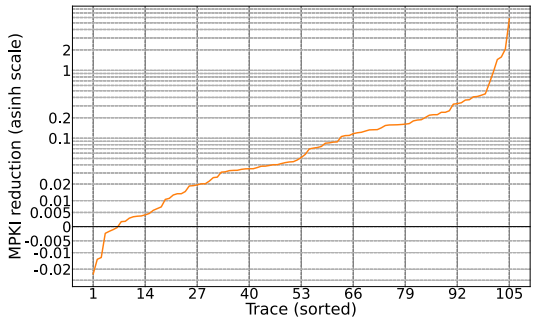

Figure 8: MPKI reduction over the TAGE-SC-L baseline.

dependencies between branch instructions and their producer instructions, capturing their correlations through pre-execution and related mechanisms [6]. The key difference between these techniques and RUNLTS is that RUNLTS directly captures correlations between register values and branch outcomes.

Heil *et al.* proposed a technique that directly feeds register-value information to the branch predictor [1]. Both their design and ours are similar in that a backing predictor makes most predictions and incorporates register-value information only for hard-to-predict branches. They implement this idea with a tagged predictor, whereas we embed it into an SC hashed perceptron predictor.

The key difference lies in the scope of register usage. Their approach is *local*; it uses the difference between two registers computed by the branch instruction being predicted. In contrast, our approach is *global*; it exploits correlations involving any register, regardless of whether that register appears in the branch instruction. This global view lets us capture correlations with registers that are loaded with different values in each loop iteration.

8 Conclusion

In this paper, we proposed RUNLTS, a branch predictor that scales to larger hardware resources by leveraging novel history-length sets, a dynamic entry allocation scheme, and a register-value correlation component. In particular, the register components, which utilize recently produced register values without explicitly tracking register dependencies, capture the correlation between register values and branch outcomes and contribute a significant improvement in accuracy. Experimental results on the CBP2025 training traces demonstrated that RUNLTS achieves significant improvements in prediction accuracy across a wide range of workloads.

Table 2: Storage budget breakdown of table predictors.

Component	Entry structure	# of entries	Storage (bits)
Bim	prediction (1), hysteresis (1/4)	2^{17}	163840
Tagged table (low banks)	useful/newly (1), ctr (3), tag (9)	9×2^{11}	239616
Tagged table (high banks)	useful/newly (1), ctr (3), tag (13)	25×2^{11}	870400
Global history			4316
Path history			27
Allocation monitoring counters	Useful (16), Decay (16)		32
U-bit monitoring counter	TICK (10)		10
Meta predictor	UseAltOnNA (5)	2^4	80
Random number generator	Seed (64)		64
Total			1278385

Table 3: Storage budget breakdown of SC components.

Component	Prediction weight table (WT)	Usefulness weight table (UT)	Auxiliary data	Storage (bits)
sB: Bias	$7 \times \left(2^{10} + 2^{10} + 2^{10}\right)$	6×2^3		21552
sG: Global backward dir	$6 \times \left(2^{10} + 2^{10} + 2^{11}\right)$	6×2^3	history (40)	24664
sP: Forward taken path	$6 \times \left(2^9 + 2^9\right)$	6×2^3	history (16)	6208
sL: 1st local	$6 \times \left(2^{10} + 2^{10} + 2^{11} + 2^{11}\right)$	6×2^3	history (18×256)	41520
sS: 2nd local	$6 \times \left(2^{10} + 2^{10} + 2^{11} + 2^{11}\right)$	6×2^3	history (21×16)	37248
sT: 3rd local	$6 \times \left(2^{10} + 2^{10} + 2^{11} + 2^{11}\right)$	6×2^3	history (19×16)	37216
sC: Call-stack	$6 \times \left(2^{10} + 2^{10} + 2^{11} + 2^{11} + 2^{11}\right)$	6×2^3	history (47×8), ptr (3)	49579
sI: IMLI	$6 \times \left(2^{10} + 2^{11}\right)$	6×2^8	BrIMLI (10), TaIMLI (11) last backward PC (64)	20117
sR: Register	$6 \times \left(2^7 + 2^8 + 2^9\right) \times 8$	$6 \times 65 \times \left(2^3 + 2^3 + 2^3\right)$	{ valid (1), payload (14) decay_ctr (8) } $\times$ 65	53863
Meta predictors			FirstH (7), SecondH (7)	14
Update thresholds			global (12), local (8×2^6)	524
Total				292505

References

[1] T.H. Heil, Z. Smith, and J.E. Smith. 1999. Improving branch predictors by correlating on data values. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. 28–37.

[2] Yasuo Ishii, Keisuke Kuroyanagi, Takeo Sawada, Mary Inaba, and Kei Hiraki. 2011. Revisiting Local History to Improve the Fused Two-Level Branch Predictor. In *Proceedings of the 3rd Championship Branch Prediction Workshop (CBP-3)*.

[3] Yasuo Ishii, Jaekyu Lee, Krishnendra Nathella, and Dam Sunwoo. 2021. Re-establishing Fetch-Directed Instruction Prefetching: An Industry Perspective. In *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 172–182.

[4] Pierre Michaud. 2018. An Alternative TAGE-like Conditional Branch Predictor. *ACM Transactions on Architecture and Code Optimization* 15, 3, Article 30 (2018).

[5] Tomoki Nakamura, Toru Koizumi, Yuya Degawa, Hidetsugu Irie, Shuichi Sakai, and Ryota Shioya. 2020. D-JOLT: Distant Jolt Prefetcher. In *First Instruction Prefetching Championship (IPC-1)*.

[6] Stephen Pruett and Yale Patt. 2021. Branch Runahead: An Alternative to Branch Prediction for Impossible to Predict Branches. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 804–815. https://doi.org/10.1145/3466752.3480053

[7] Alberto Ros and Alexandra Jimborean. 2021. A Cost-Effective Entangling Prefetcher for Instructions. In *Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA)*. 99–111.

[8] André Seznec. 2014. TAGE-SC-L Branch Predictors. In *Proceedings of the 4th Championship Branch Prediction Workshop (CBP-4)*.

[9] André Seznec. 2016. TAGE-SC-L Branch Predictors Again. In *Proceedings of the 5th Championship Branch Prediction Workshop (CBP-5)*.

[10] André Seznec. 2020. The FNL+MMA Instruction Cache Prefetcher. In *First Instruction Prefetching Championship (IPC-1)*.

[11] André Seznec. 2024. TAGE: an engineering cookbook. HAL Inria, RR-9561. https://hal.science/hal-04804900 Submitted on December 4, 2024; Last modified March 28, 2025.

[12] André Seznec and Pierre Michaud. 2006. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-level Parallelism* 8 (2006), 1–23.

[13] Rami Sheikh and Saransh Jain. 2025. ramisheikh/cbp2025: Championship Branch Prediction 2025. https://github.com/ramisheikh/cbp2025. Accessed 2025-06-07.

[14] Ryota Shioya. 2016–2023. Konata: Instruction Pipeline Visualizer for Onikiri2-Kanata and Gem5-O3PipeView. https://github.com/shioyadan/Konata. Accessed 2025-05-02.

[15] Zichao Xie, Dong Tong, and Xu Cheng. 2013. An energy-efficient branch prediction technique via global-history noise reduction. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design (ISLPED)*. 211–216.

A Cost Analysis

Table 2 and 3 summarize the storage costs of RUNLTS. The table predictor consumes 1278385 bits (156.05 KiB), and the SC predictor consumes 292505 bits (35.71 KiB). The total storage cost is 1570890 bits (191.75 KiB), compliant with the CBP2025 rules.