TAGE-SC-L with a Code Structure Correlator

Lingzhe(Chester) Cai
chestercai@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Aniket Deshmukh
a.deshmukh@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Evan Lai
laievan@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Ali Mansoorshahi
kayvan.mansoor@gmail.com
University of Texas at Austin
Austin, Texas, USA

Mircea Tatulescu
mirceatatulescu@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Isaac Nudelman
isaac.nudelman@utexas.edu
University of Texas at Austin
Austin, Texas, USA

Yale N. Patt
patt@ece.utexas.edu
University of Texas at Austin
Austin, Texas, USA

Abstract

TAGE-SC-L is the best known branch predictor to date. Despite its storage efficiency, a large amount of storage must be dedicated to easy to predict branches. We propose the Code Structure Correlator, an O-GEHL-style predictor that relieves TAGE-SC-L capacity. Our final tuned TAGE-SC-L achieves (3.468 MPKI) and with the addition of CSC achieves (0.04 MPKI reduction). Furthermore, our predictor significantly reduces the capacity pressure on the lower histories, allowing TAGE to focus on the high histories.

1 Introduction

Despite decades of research, branch prediction continues to limit single-thread performance. The state of the art, TAGE-SC-L [6, 7], has proven to be an effective branch predictor design in both academia and in real commercial products. However, TAGE-SC-L still suffers from capacity problems, especially as code and branch footprints continue to grow in modern workloads [1]. While these branches are often easy-to-predict, they still take up valuable storage in TAGE-SC-L, leaving less room for the hard-to-predict branches.

We present a hybrid predictor composed of two components: a TAGE-SC-L, and our O-GEHL-style Code Structure Correlator (CSC). This allows TAGE to allocate more storage to the longer history tables, and deal with hard-to-predict branches. Our submission includes the following:

- CSC - An O-GEHL-style predictor predicting based on code structure.
- A method for fuzzing TAGE-SC-L to find new configurations.

2 Related Work

2.1 O-GEHL

Optimized Geometric History Length (O-GEHL) is another history-based predictor [3]. It takes different histories and uses them to index tables. Counters are read out from these tables and then summed together to give a final prediction.

'CBP 2025', June 21, 2025, Tokyo, Japan
2025. ACM ISBN How.to.remove.this?(Not.ACM)
https://doi.org/How.to.remove.this?(Not.ACM)

2.2 TAGE-SC-L

Our main predictor is the TAGE-SC-L, which is composed of 3 components: TAGE, a Statistical Corrector and a Loop Predictor. TAGE, the main component in TAGE-SC-L, was originally introduced in 2006 [8], and improved in [6] and [7]. TAGE is composed of an untagged bimodal table along with several tagged tables, where each tagged table corresponds to a history length chosen from a geometric sequence. The tagged tables are indexed by hashing the PC and the history at the corresponding length. A final prediction is picked between the highest matching tag and the second highest matching tag. Note that TAGE improves upon prior work with tags to reduce aliasing and uses muxes to pick the final prediction as opposed to computing the sum for the final prediction.

Alongside TAGE is SC, the statistical corrector, an O-GEHL style predictor first introduced in 2011 [5]. The SC uses the PC and local histories as inputs and can override TAGE when local history is more correlated with a particular branch.

Finally, there is L, the loop predictor, a local history based predictor introduced in the 2nd CBP competition in 2007 [4]. L has minimal contribution to the overall performance and was kept due to its small storage footprint.

3 High-Level Design Overview

Our predictor is a hybrid predictor composed of two parts: TAGE-SC-L and CSC. The overview can be seen in Figure 1. TAGE-SC-L, the bloom filter, and CSC are looked up in parallel. The bloom filter is used to determine if the prediction should come from TAGE or CSC.

4 Predictor Operation

4.1 TAGE-SC-L Again, Again

The main component of our predictor remains the TAGE-SC-L predictor, which was adapted to the size constraints of this competition. In order to accomplish this, we tuned parameters using a fuzzing framework guided by heuristics and an evolutionary approach. At the core is a fuzzer implemented using DEAP [2]. Each candidate configuration, termed an "individual", encodes predictor parameters such as the number of low and high history banks, associativity

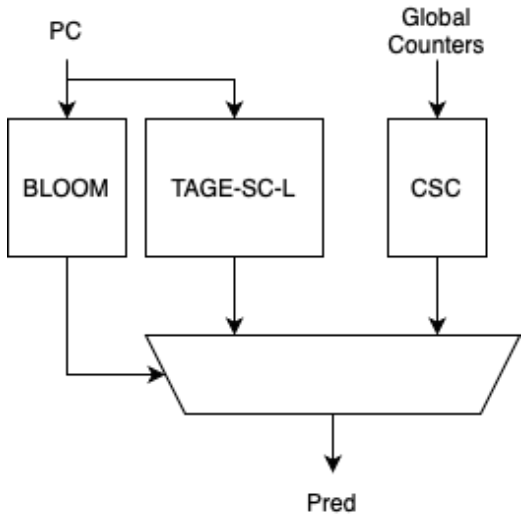

Figure 1: Overview of the hybrid predictor

```
def jitted_function(a, b):
    if a is String:
        bail(this)
    if b is String:
        bail(this)
    if a is Function:
        bail(this)
    if b is Function:
        bail(this)
    if a is HiddenClass:
        bail(this)
    ...
    // a, b are int32
    return a + b
```

Figure 2: Example JITed Function

boundaries, table sizes, counter precisions, and statistical corrector component widths. The complexity of interactions among these parameters necessitated heuristics capable of guiding the exploration process towards correct, reasonable configurations.

Our heuristics were derived directly from analyses of the TAGE-SC-L predictor architecture. Given the observation that larger configurations are almost always better than smaller ones, we decided to only explore configurations that were within 5% of the target size of 192 KB. To do this, we repeatedly fuzz configurations until they are within the target range. To ensure validity, each individual was passed through a repair routine that enforced correctness constraints—for example, proper alignment of associativity boundaries, consistency between NHIST and BORN, and minimum bank counts required to support the chosen history table layout.

During the repair routine, an individual may exceed the size constraint. If this happens, we apply a series of weighted random reductions to reduce the predictor size. Each potential modification—such as shrinking bank counts, narrowing associativity ranges, or reducing statistical corrector precision—is assigned a weight based on its estimated size impact. Higher impact changes, such as decreasing the log size of banks, are given a smaller weight, while lower impact changes, such as adding another bank, are given a larger weight. The predictor is then randomly shrunken according to these weights. The idea is to shrink the configuration back under 192 KB, while still keeping it as close as possible. The randomness is to reach higher coverage while fuzzing. This weighting system helped steer the search toward compact yet structurally reasonable configurations and formed the core of our tuning strategy.

Alongside adapting the TAGE-SC-L parameters to fit within the size constraints, we made a slight modification to the allocation policy. Given the larger predictor size, an extra entry is allocated when trying to allocate from a lower history (HitBank < 18). This allows branches that would benefit from having a longer history to get there faster without any additional hardware.

4.2 Code Structure Correlator

In addition to the TAGE-SC-L, we add the Code Structure Correlator (CSC). The CSC is a O-GEHL-style predictor [3]. The CSC attempts to use code structure signals to predict the outcome of branches without using the PC of the branch. This could be used in a number of cases, including code duplication, dynamic relocation of code snippet, cold branches and exception checking branches. As a motivating example, consider code generated by a JavaScript JIT as shown in Figure 2. The entry point of such JITed code will contain a series of branches in rapid succession to ensure that assumptions gathered before JITing are still valid. These guard branches are extremely likely to be not-taken, as any taken branch will result in the JITed function being thrown out (and overwritten by a stack frame from the interpreter). The CSC aims to generalize patterns like this.

In order to capture the code structure, we use the features described in the following sections. The overall design can be seen in Figure 3. When making a prediction, the features are used to index their own tables. Each entry in the table contains an n-bit saturating counter. We experimentally found that 9 was a good width for these counters. The counters are then summed together via an adder tree, and if the result is positive, the prediction is taken; otherwise, it is not taken.

On every branch resolution, we update the weights in the table. If the direction is taken then all the relevant counters are incremented, otherwise, the counters are decremented. Note that even if a branch is predicted by TAGE-SC-L, it still updates the counters in the CSC.

4.2.1 Call Depth. This is a counter that is incremented on call instructions and decremented on return instructions. This allows the CSC to detect when we are deep in the call stack.

4.2.2 Global Bias. This is a saturating 3-bit counter that tracks the overall bias direction of conditional branches. The intuition behind this is that type checks, similar to those shown in figure 2, all go the same direction.

4.2.3 Far Branch Count. Counter that tracks the number of branches with an target address more than 4K away within 100 and 1000

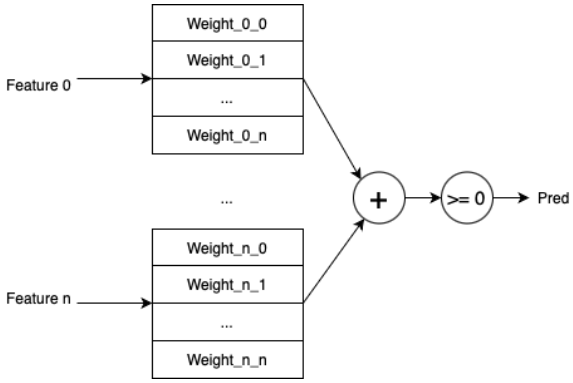

Figure 3: Overview of the CSC

instruction windows. This can be implemented through a bit vector, each bit representing if an instruction was a far branch.

4.2.4 Indirect Branch Count. This is the same as the "Distant Branch Count" in section 4.2.3 except we count indirect branches instead. This is also tracked for windows of 100, and 1000 instructions.

4.3 Combining the Two Predictors

We use a bloom filter as a selector to pick between the prediction from TAGE-SC-L and CSC. If a branch's PC hits in the bloom filter, we pick the prediction from TAGE-SC-L, otherwise we pick the prediction from CSC. The bloom filter starts out empty, meaning all branches starts being predicted by the CSC. Once there is a misprediction from the CSC at the time of predictor update, the branch's PC is inserted into the bloom filter. The idea is that, after a single misprediction from the branch, all other instances of the branch will be predicted by TAGE.

At update time, all branches updates the CSC. The TAGE-SC-L predictor is only updated if the branch PC is hit in the bloom filter. This allows branches that are easily predictable to be predicted by the CSC, avoiding the use of TAGE storage.

5 Experimental Results and Analysis

Comparing to the baseline 64 KB TAGE-SC-L with a 3.75 MPKI, our predictor achieves a 3.46 MPKI. The majority of the saving comes from the increased capacity of the in the TAGE higher histories. The MPKI and wrongpath cycles PKI are shown in the table below.

	web	media	compress
BrMisPKI	3.3030	0.8564	2.7342
CycWpPKI	110.6938	28.5441	84.1648
	int	fp	infra
BrMisPKI	4.2518	4.1011	2.3921
CycWpPKI	164.0265	164.9199	208.3353

6 Discussion

6.1 CSC

Since the CSC doesn't rely on a specific branch for generating any predictions, it shares the prediction information of multiple branches with the same counter. Comparing to TAGE-SC-L that indexes explicitly based on the PC of the branch, our CSC implementation explicitly uses the common code structure for predicting branches, thus allowing the predictor to be much more efficient at predicting easy to predict branches. In addition, CSC is able to predict cold/infrequent branches, given that there is some correlation with the signals that we collect. Overall, by only training TAGE-SC-L with branches that our CSC cannot deal with, this allows TAGE to save space in the low history tables. We noticed this empirically as our TAGE-SC-L tuning algorithm produces configurations with much smaller storage (7.5K) dedicated to the low histories when CSC is enabled.

CSC was implemented as a separate component, and not as a part of SC for two reasons. First, existing components inside of SC use the current branch PC as part of the index computation, which can greatly interfere with CSC. The sum of the weights from existing components can easily outweigh the CSC weights, even when they could be just noise, since the branch has not been observed. Second, building CSC as a secondary component allows a custom design of the selection logic independent of the signals inside of TAGE and SC.

6.2 CSC - Feature sensitivity

In order to determine the feature sensitivity, we performed a simple ablation study. The results of that can be seen in Table 1. Call depth is the most important feature. We believe this is due to deeper calls being shared code that has lots of easy to predict input validation style branches.

Feature Name	MPKI increase
Call Depth	0.0918
Global Bias	0.0248
Indirect within 100 insts	0.0056
Indirect within 1000 insts	0.0043
Far Br within 100 insts	0.0033
Far Br within 1000 insts	0.0021

Table 1: MPKI Increase if Each Feature Was Turned off

6.3 Bloom Filter

An oracle implementation of the bloom filter can only provide 0.002 MPKI improvement over our realistic bloom implementation. On a realistic processor, the cost of the bloom filter may be reduced or even completely eliminated. For example, one can choose to implement this as a bit in the BTB entry, refering to whether the branch should be predicted by the CSC or TAGE-SC-L.

The bloom filter should be cleared periodically in a realistic implementation. This can be done either after a fix number of instruction or at a context switch.

7 Conclusion

TAGE-SC-L is a highly accurate predictor for complex branches, but it must allocate space for simple branches. TAGE-SC-L's accuracy can be improved by using a side predictor to save predictor storage. We have described CSC which uses code structure metrics to make predictions. This improves the performance of TAGE-SC-L, and

overall. CSC can better handle the increased footprint of modern workloads without taking up much storage inside of TAGE.

References

[1] 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. https://doi.org/10.1109/ISPASS51385.2021.00028

[2] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research* 13, 1 (2012), 2171–2175.

[3] Andre Seznec. 2004. The o-gehl branch predictor. *The 1st JILP Championship Branch Prediction Competition (CBP-1)* (2004).

[4] 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.

[5] André Seznec. 2011. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 117–127.

[6] André Seznec. 2014. TAGE-SC-L Branch Predictors. In *Proceedings of the 4th Championship Branch Prediction*. Minneapolis, United States. https://inria.hal.science/hal-01086920

[7] 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

[8] André Seznec and Pierre Michaud. 2006. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction-Level Parallelism* 8 (2006), 23.

A Cost Analysis

The following table presents a breakdown of predictor storage, with the the total footprint remains within the 192 KB budget. Because we are using a single bloom filter, we were able to size our predictor to exactly 192 KB. Note that the SC modifications were omitted in the table for brevity—here is a list of the changes:

- **`PERCWIDTH`**: Increased from 6 to 8.
- **`LOGBIAS`**: Increased from 8 to 9.
- **`LOGLNB`**: Increased from 10 to 11.

Component	Details	Cost
TAGE-Low History Banks	5 banks · 2^{10} entries of size (3+1+8)	7.5 KB
TAGE-High History Banks	18 banks · 2^{12} entries of size (3+1+12)	144 KB
TAGE-Base Predictor	2^{15} entries + 2^{13} hysteresis	5 KB
TAGE-Auxillary	UseAlt, History, etc... (unchanged from 2016)	0.38 KB
Statistical Corrector	Minor resizing from 2016 (for brevity, omitted)	11.95 KB
Loop Predictor	Unchanged from 2016	0.15 KB
CSC-Predictor	6 features ·2^8 entries of size 9	1.69 KB
CSC-Bloom Filter	Sized to fill up the remaining budget	21.33 KB
TOTAL		192 KB

B Prediction History Checkpoints

In addition to the checkpoints necessary to support the baseline TAGE-SC-L implementation, we need to checkpoint the inputs for the CSC predictor at the time of the prediction. These inputs are 7 bit wide each. Since our implementation includes 6 features, this would be an additional 42 bits per branch.

To track the number of far branches and indirect branches within a window, two 1000 bit vectors are used. Checkpointing the 2 bit vector for each branch can be very expensive. Alternatively, we can simply increase the length of the vector to support rollback on a misprediction. This would require increasing the bit vectors length by the maximum number of instructions in the pipeline.