Exploration of Compressed Branch Histories on Dynamic TAGE Implementations

Submitted by

Ayan Deep Hazra

Kiranmay Sekar

Pritheshwar Thirugnanasambantham

Tingbiao Li

Department of Electrical and Computer Engineering

UNIVERSITY OF WISCONSIN–MADISON

December 2023

Table of Contents

CHAPTER 1
INTRODUCTION

1.1 Motivation to do the project

The pursuit of efficient computer architecture has placed branch prediction at the forefront of processor performance optimization. Within this realm, the Tagged Geometric History Length (TAGE) branch predictor has been a notable benchmark since its introduction, especially for its use of data compression principles derived from partial matching algorithms. This efficient approach has minimized pipeline stalls and maximized instruction throughput more effectively than previous models.

Despite TAGE's prominence, its performance is capped by the static allocation of its table sizes, which fails to accommodate the diverse and dynamic nature of software workloads. Our exploration into this domain underscores the need for a system that can adapt to varying workload demands, particularly those requiring longer history branches for accurate prediction.

Our project endeavors to reproduce the dynamic branch prediction capabilities of TAGE [3], incorporating advancements from recent research. We are not innovating the reconfigurable architecture; instead, we are replicating the existing dynamic TAGE branch predictor within the gem5 simulation environment. Our objective is to verify the effectiveness of this established method.

1.2 Objectives of the project

To streamline our project and make efficient progress, we decided to divide our efforts into two groups:

- **Gem5 Team :** This group primarily focused on software-based simulations using the Gem5 tool, such as -
  - *Gem5 Setup*: Configure and set up Gem5 environments for simulating the Dynamic TAGE branch predictor
  - *Experiment Design*: Design a comprehensive set of experiments to evaluate dynamic TAGE's performance and accuracy under various SPEC workloads.
  - *Data Collection*: Execute simulations, collect data, and analyze the results to assess dynamic TAGE's behavior.
  - *Parameter Tuning*: Fine-tune dynamic TAGE's configuration parameters to optimize its prediction accuracy.

- **Hardware Implementation Team:** This group was responsible for the hardware-level implementation of the TAGE branch prediction algorithm in RTL.
  - *RTL Design*: Develop the hardware architecture for dynamic TAGE
  - *Synthesis and Verification*: Use hardware synthesis tools to convert RTL code into hardware descriptions and rigorously verify the correctness of the design.
  - *Testbench Creation*: Create testbenches for RTL simulations to validate the hardware implementation's functionality.
  - *Performance Evaluation*: Analyze the hardware implementation in terms of resource utilization, power efficiency, and potential speedup in real processor architectures.

CHAPTER 2

BACKGROUND THEORY

2.1 Dynamic TAGE

Dynamically sizing the TAGE Branch Predictor, [3] proposes the following for a dynamic implementation of the TAGE predictor

- A set of geometrically increasing histories (similar to that in conventional TAGE)
- A set of 32 tiles, each with 64 pattern history entries
- Interconnects to group the tiles into dynamically configurable pattern history tables
- Quantum : defined as the time taken to accumulate 100,000 predictions
- A scoring unit to determine the size of the tables for the next quantum
- A configuration vector that is used to set the size of the tables and configure the output

While TAGE's design has catered to branches with short histories, this is not a one-size-fits-all solution. Benchmarks often require attention to longer history branches, which are under served by the static allocation of storage in TAGE's design. The allocation of storage to short history branches without considering the demands of longer histories increases misses per kilo-instruction (MPKI), leading to sub-optimal performance. A reconfigurable architecture that dynamically allocates storage to the history lengths most in demand during runtime. This solution involves partitioning the runtime into quanta and reallocating storage at the beginning of each quantum based on the collected data from prior quanta.

2.2 Other Related Work

BADGR [1] raises concern about the check-pointing technique used for the Global History Register (GHR) in TAGE predictors. In TAGE, the GHR's size can be up to 2000 bits. This means that updating the GHR is expensive in and of itself, but reverting to a check-pointed state after a speculative update of the GHR can be even more expensive. Commonly used check-pointing techniques such as Backward Shifting or Circular Buffers are not ideal as they have high dynamic power consumption or high overhead for a GHR read, respectively. In a modern Out-of-Order processor, the corruption of the GHR poses a significant threat to performance, as branch predictors are integral to exploiting Instruction Level Parallelism (ILP). Thus, [1] proposes the BADGR check-pointing mechanism for TAGE, which reduces dynamic power consumption and complexity of the logic for GHR reads.

CHAPTER 3

METHODOLOGY

In the Methodology section of our study, we commence by offering a detailed exposition of each novel component within the dynamic TAGE architecture. This comprehensive breakdown not only elucidates the functional intricacies of each element but also clarifies their roles in achieving the dynamic behavior of the system. Subsequently, we delineate the process by which this innovative architecture has been implemented within the gem5 simulation environment and RTL simulation.

3.1 The Architecture

The new architecture is depicted in Figure 3.1 (page 5), which not only incorporates the traditional TAGE structure but also introduces several key components designed to dynamically adjust the table sizes. Firstly, a set of tiles along with two Reconfigurable Interconnects are responsible for grouping tiles into tables. Additionally, there is a scoring unit tasked with collecting runtime information during each runtime quantum. This information is critical for determining the size of all tables for the subsequent quantum. Lastly, the configuration vectors, designated by the scoring unit at the end of each quantum, are utilized by the Reconfigurable Interconnects to adjust the table sizes at the commencement of the next quantum. These enhancements are pivotal for the dynamic resizing capability that defines this new architecture.

3.1.1 Tables, Tiles, and Configuration Vectors

Regarding the number of tables, we have choice 16, drawing from a set of 32 tiles to allocate storage for each table, with each tile having 64 entries. These entries are the same as TAGE architecture, with each one comprising three fields: tag, prediction counter, and useful bit. And the tiles are organized as direct mapped tables. As for the Configuration Vector, we represent the tile-to-table allocation using a 16-bit digital, where each bit d_i corresponds to the number of tiles allocated to table i, and we constrain it to be either 0 or a power of 2. This ensures that even if a table is assigned zero tiles, we still maintain 16 history registers. If we determine that any entry is not useful, then we only allocate tiles to those histories that we identify as useful at the start of each runtime quantum. This strategic allocation maximizes the efficiency of our tables by dynamically adjusting to the utility of the histories, ensuring that resources are not wasted on unproductive entries.

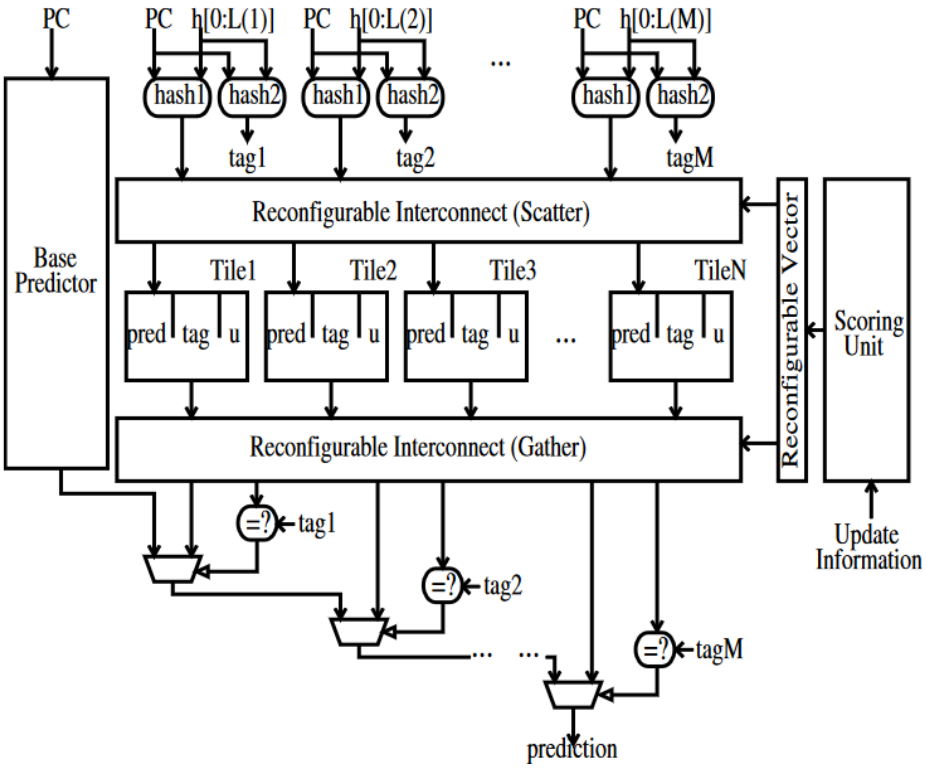

Figure 3.1: Block Diagram of the Reconfigurable TAGE Architecture [3]

3.1.2 The Reconfigurable Interconnect

The apparatus incorporates two *Reconfigurable Interconnects*. The first connects the history registers to the tiles, tasked with mapping each history to the appropriate tile. The second reconnect links the tiles to the output multiplexer, orchestrating the mapping of each tile back to its corresponding history register. Figure 3.2 (page 6) demonstrates a configuration with two history registers and four tiles.

In this diagram, each 'x' represents a switch that can be activated or deactivated by the Configuration Vector. The hash function plays a critical role here, generating an index wherein the lower bits are employed to search an entry and the upper bits are dedicated to tile selection. The concluding segment of the Reconfigurable Interconnect is a comparator, which juxtaposes the upper bits of the hash function's index against the tile ID of a tile, and the lower bits against the entry index, culminating in the target output. The tile ID, as dictated by the Configuration Vector, denotes the index of that specific tile within its respective table. This design ensures a meticulous and dynamic mapping process, vital for the precision of the branch prediction mechanism.

3.1.3 The Scoring Unit

The Scoring Unit produces the Configuration Vector for the next quantum. It separates the run-time into two phases: a learning phase (First seven quantum for the run-time quanta) and an

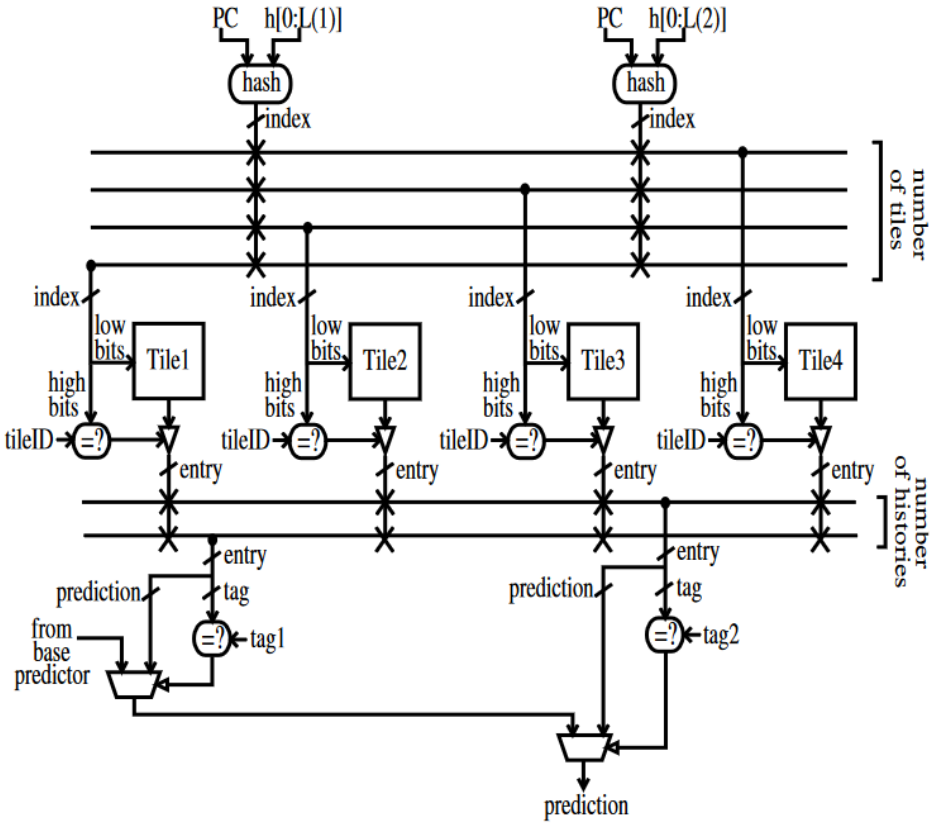

Figure 3.2: The Reconfigurable Interconnect [3]

adaptive phase (The rest quantum for the run-time quanta). Each quantum is the time it takes to accumulate 100,000 predictions.

1. **Learning Phase**
   The learning phase is integral to the dynamic optimization of the program, with the objective of generating seven configurations that will be utilized for the program's adaptive phase. This phase spans seven quanta, with each quantum initiating with a distribution of tiles to tables as dictated by a Configuration Vector. The initial vector is uniform, with the allocation being 2 for each of the 16 slots: 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2. Throughout each quantum, key metrics are collected, including attempted allocations, conflicts, and mispredictions. Upon the completion of a quantum, this data is processed by the scoring unit to derive a new Configuration Vector for the next quantum. Additionally, the misprediction count and the current Configuration Vector are archived for future reference in the adaptive phase of the program. This process ensures that the program adapts to the behavior observed during the learning phase, optimizing future performance.

2. **Adaptive Phase**
   During the adaptive phase, the system employs a dynamic strategy to enhance prediction accuracy by selecting the most effective Configuration Vector for each quantum. This selection is guided by the number of mispredictions recorded for each of the seven con-

figurations established in the learning phase. At the conclusion of each quantum in the adaptive phase, the misprediction count will be updated for the configuration that was in use. Unless a particular reason suggested a different approach, the system chose the Configuration Vector for the upcoming quantum based on the one with the fewest misprediction, and the process is repeated until the program terminates. To minimize the cost of switching configurations, a 2-bit counter is incorporated to prevent frequent changes. The counter will reset to 0 if the system switches to a new configuration and it will increase if the system keeps the same configuration. A switch to another configuration, prompted by a lower misprediction vector entry, causes the counter to decrement. A configuration change is only executed when the counter reaches zero, ensuring stability and preventing erratic switches that might result from an anomalous quantum.

The most important thing in the scoring unit is the reallocation algorithm. The reallocation algorithm presented involves a two-step process for optimizing table storage in a dynamic system. Initially, the algorithm collects underutilized tiles from tables and then redistributes them as needed. Tables contribute tiles if the rounded-up count of their non-zero useful bit entries is less than their current size. Tiles are then reassigned to the most congested tables, assessed by two metrics: attempted allocations and conflicts experienced. The conflict algorithm redistributes tiles to tables that exceed the average number of conflicts, while the attempt algorithm favors tables with above-average allocation attempts and those with longer histories. The hybrid algorithm introduced combines the conflict and attempt algorithms, choosing between them based on misprediction rates. If mispredictions are frequent, the conflict algorithm is chosen; if infrequent, the attempt algorithm is preferred. No reconfiguration occurs if mispredictions are extremely rare, to avoid unnecessary costs.

3.2 *gem5* Implementation

3.2.1 DynamicTAGE Class

The most important structures of the ***gem5*** simulator, for our project, is shown in Figure 3.3 (page 8).

The TAGE class, derived from the *BranchPredictor* class is a wrapper that calls functions of a specific TAGE implementation, using a pointer of the type *TAGEBase*. We have extended this *TAGEBase* class to create a *DynamicTAGE* derived class.

The specific TAGE implementations also contain a dynamically created geometric table (gtable), whose parameters can be set using their respective Python files. The *gtable* is structured, in existing TAGE implementations, as shown in Figure 3.4 (page 8).

An important point is to be noted here - *gem5* doesn't seek to replicate the exact hardware structures in a CPU. Instead, the logic of the design is implemented at a very high level of abstraction. Thus, in Figure 3.4 (page 8), we see that the *gtable* is indexed using a simple

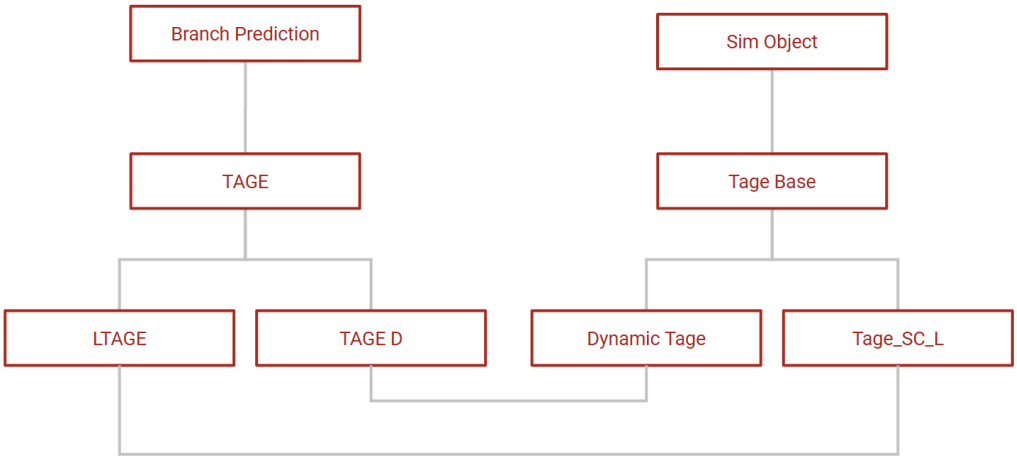

Figure 3.3: Structures Important to Our Project in gem5

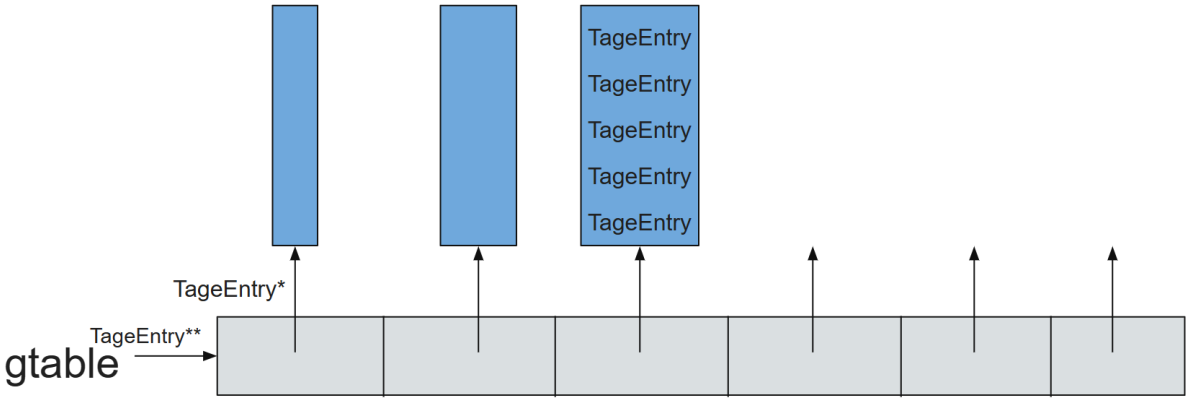

Figure 3.4: Existing TAGEBase Class's *gtable* Implementation

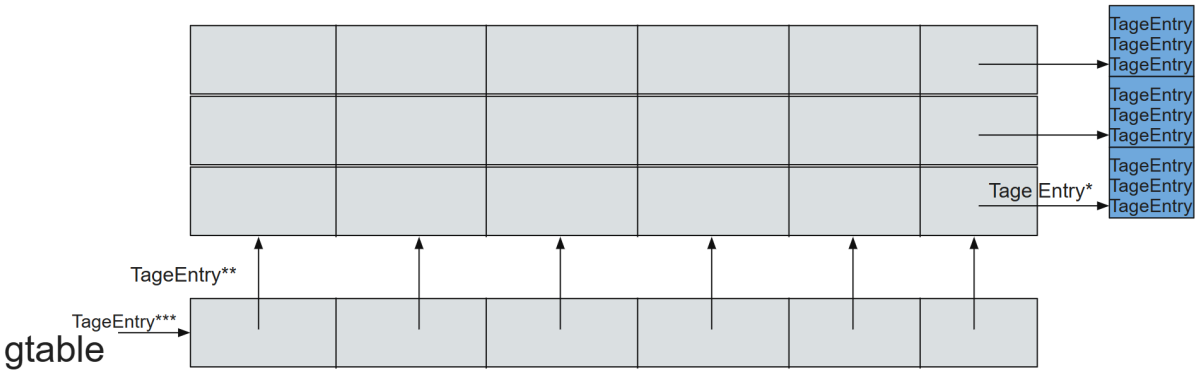

Figure 3.5: DynamicTAGE Class's *gtable* Implementation

```
//Look for the bank with longest matching history
for (int i = nHistoryTables; i > 0; i--) {
    currentTableIndex = bi->tableIndices[i];
    tileIndex = calcTileIndex(currentTableIndex);
    indexWithinTile = calcIndexWithinTile(currentTableIndex);
    DPRINTF(Tage, "\nLook for the bank with longest matching history");
    if (noSkip[i] && gtable[i][tileIndex][indexWithinTile].tag == tableTags[i]) {
        bi->hitBank = i;
        bi->hitTile = tileIndex;
        bi->hitBankIndex = tableIndices[bi->hitBank];
        break;
    }

}
```

Figure 3.6: Simple Overriding Changes to Functions of TAGEBase in DynamicTAGE

integer index, where the entries are of the type *struct TageEntry*.

Following the above philosophy, we arrive at a simple extension of the *gtable*'s structure. The new *gtable* structure for the *DynamicTAGE* class is shown in Figure 3.5 (page 9).

Here, we see that *gtable* is now a 2D array of *TageEntry* pointers. Each row mimics the behaviour of a tile. The deallocation and allocation of tiles to a particular history table, is a simple allocation and deallocation of memory at C++ level. There is no need to reproduce the hardware interconnect mentioned in the paper [].

Apart from the re-assignment of tiles feature in Dynamic TAGE, it behaves very similar to the typical TAGE implementation. The extension of the TAGEBase class then, involved simple changes as shown in Figure 3.6 (page 9).

3.2.2 Other Related Structural Changes

Other code changes have been highlighted from Figure 3.7 (page 10) to Figure 3.13 (page 13).

```
branchPred = Param.BranchPredictor(
    TAGE(numThreads=Parent.numThreads), "Branch Predictor"
)
```

Figure 3.7: BaseO3CPU.py

```
class DynamicTAGE(TAGEBase):
    type = "DynamicTAGE"
    cxx_class = "gem5::branch_prediction::DynamicTAGE"
    cxx_header = "cpu/pred/dynamic_tage.hh"

    entries_per_tile = Param.Unsigned(64, "Number of Entries Per Tile")
    no_of_tiles = Param.Unsigned(21, "Total Number of Tiles")
    current_configuration_vector = VectorParam.Int(
        [13, 4, 4, 4, 4, 4, 4, 4], " Static Config Vector for Testing"
    )

    current_config = VectorParam.Int(
        [13, 0, 0, 0, 0, 0, 0, 0], " Static Current Config for Testing"
    )
```

Figure 3.8: BranchPredictor.py - DynamicTAGE SimObject

```
class TAGE(BranchPredictor):
    type = "TAGE"
    cxx_class = "gem5::branch_prediction::TAGE"
    cxx_header = "cpu/pred/tage.hh"

    tage = Param.TAGEBase(DynamicTAGE(), "Tage object")
```

Figure 3.9: BranchPredictor.py - Default Predictor Set to DynamicTAGE

filespace > k > ksekar2 > ece752 > gem5 > src > cpu > pred > ConfigurationVector.hh

```hh
#pragma once

#include <vector>
#include <string>

class ConfigurationVector {
public:
    // Constructor that initializes the configuration from a string
    explicit ConfigurationVector(const std::string& config);

    // Get the number of tiles allocated for a specific table
    int getTilesForTable(int tableIndex) const;

    // Update the configuration vector dynamically
    void updateConfiguration(const std::string& newConfig);

private:
    // Stores the number of tiles allocated to each table
    std::vector<int> tableAllocations;

    // Helper method to parse the configuration string
    void parseConfig(const std::string& config);
};

```

Figure 3.10: config vector.hh - Configuration Vector Class

filespace > k > ksekar2 > ece752 > gem5 > src > cpu > pred > ConfigurationVector.cc

```cc
#include "ConfigurationVector.hh"

ConfigurationVector::ConfigurationVector(const std::string& config) {
    parseConfig(config);
}

int ConfigurationVector::getTilesForTable(int tableIndex) const {
    if (tableIndex >= 0 && tableIndex < tableAllocations.size()) {
        return tableAllocations[tableIndex];
    } else {
        // Handle invalid table index, could also throw an exception
        return 0;
    }
}

void ConfigurationVector::updateConfiguration(const std::string& newConfig) {
    parseConfig(newConfig);
}

void ConfigurationVector::parseConfig(const std::string& config) {
    tableAllocations.clear();
    for (char digit : config) {
        if (isdigit(digit)) {
            int tiles = digit - '0'; // Convert character to integer
            tableAllocations.push_back(tiles);
        } else {
            // Handle non-digit characters, could also throw an exception
        }
    }
}
```

Figure 3.11: config vector.cc - Configuration Vector Class Functions

filespace > k > ksekar2 > ece752 > gem5 > src > cpu > pred > scoring_unit.cc

```cpp
#include "scoring_unit.hh"

scoring_unit::scoring_unit() : totalPredictions(0), correctPredictions(0) {}

// Scoring unit 1-> Learning Phase algorithm
void scoring_unit::updatePredictionOutcome(bool wasCorrect) {
    totalPredictions++;
    if (wasCorrect) {
        correctPredictions++;
    }
}

double scoring_unit::getCurrentScore() const {
    return calculateScoreMetric();
}

void scoring_unit::suggestNewConfiguration(std::vector<int>& configVector) {

    double score = getCurrentScore();

    // check score at actual run below 0.8 threshold
    if (score < 0.8) {

        for (int &tiles : configVector) {
            tiles = (tiles > 1) ? tiles - 1 : tiles;
        }
    }


}

void scoring_unit::reset() {
    totalPredictions = 0;
    correctPredictions = 0;
}

double scoring_unit::calculateScoreMetric() const {
    if (totalPredictions == 0) return 0.0;
     return static_cast<double>(correctPredictions) / totalPredictions;
 }
```

Figure 3.12: scoring unit.hh - Scoring Unit Class

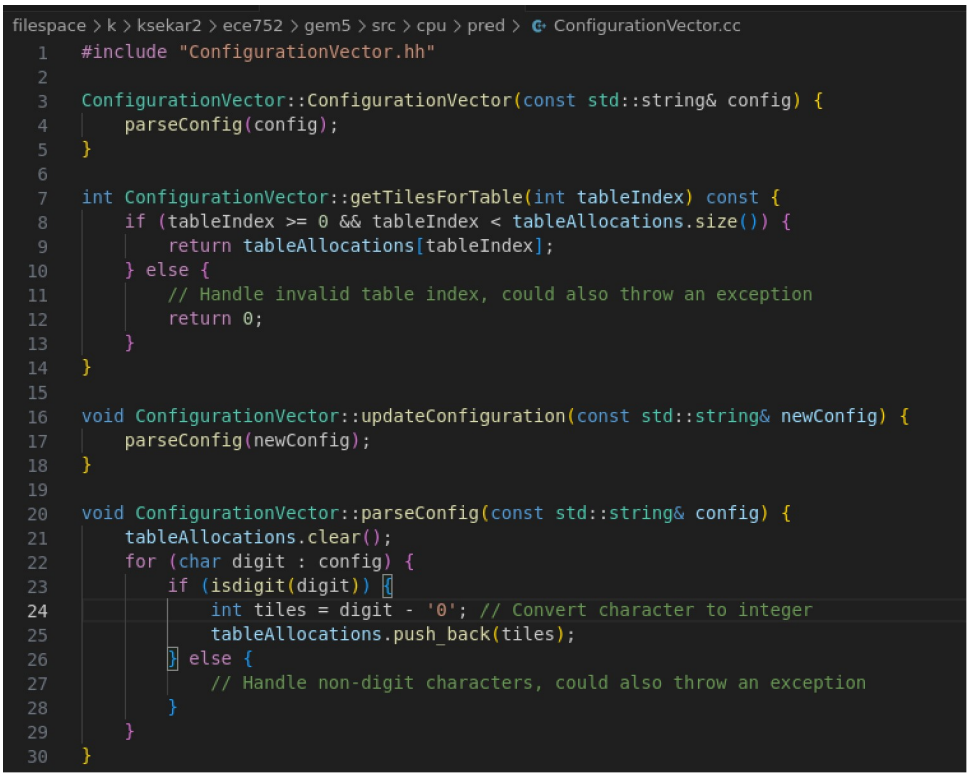

Figure 3.13: scoring unit.cc - Scoring Unit Class Functions

3.2.3 Trace Simulation

In our project we made use of the code provided from the 5th Championship Branch Prediction competition held in 2016 (CBP-16). An augmented simulator is created here for studying modern branch prediction units (BPUs).The two main components of this work are the programs simnlog and simpython. We ran simnlog program from the cbp16sim directory which were compiled to run the *TAGE-SC-L BPU* for trace *SHORT MOBILE-2.bt9.trace.gz*, *SHORT MOBILE-43.bt1.trace.gz* and *SHORT_SERVER-283.bt9.trace.gz*.

It generated a MPKI plot for each traces for the default predictor and TAGE_D predictor. Setting python variable for TAGE_D.py

```
1 $PYTHONPATH=/simpython/cbp2016.eval/traces/LONG_SERVER-1.bt9.trace.gz
  ↪ TAGE_D.py
2 $PYTHONPATH=/simpython/cbp2016.eval/traces/SHORT\_SERVER-283.bt9.trace.gz
  ↪ TAGE\_D.py
```

To create our branchpredictor, we inherited the BASEPREDICTOR abstract base class located in *cbp16sim/src/simpython/predictor.py*. We studied the implementation of the methods in the *dummy_predictor.py* file in the same directory. Also we renamed our class such that it inherits from BASEPREDICTOR a special name: TAGE_D.

```
55      for(i = 0; i < NUM_TABLES; i++) begin
56          assign scatter_index[i] = pc[0:2] ^ h[i][0:2];
57          assign tag[i] = pc ^ h[i];
58      end
```

Figure 3.14: Hashing Functionality

Results

As mentioned in the paper we observed a value close to 5.4 MPKI for 8KB budget configuration for above mentioned traces.

3.3 RTL Implementation

The RTL implementation looks at a small subset of the TAGE design. It does not implement the scoring unit, instead looking to hardcode the configuration vector and history vector values, shown in Figure 3.14 (page 14) that are provided by the scoring unit. To keep our design simple to code and verify, we utilize a structure of 2 history tables and 4 total tiles.

3.3.0.1 Hashing functionality

For both the scatter index hash and the tag bit for the line in the history table, we opt for a basic XOR hash, the idea being that both hashes will not coincide due to the reconfiguration mechanism. It is shown in Figure 3.14 (page 14)

3.3.0.2 Input to the Tiles

In the input mechanism, we copy the index over, and then set a buffer enable for all the 4 tiles, to choose if its output is needed in a particular quantum. It is shown in Figure 3.15 (page 15)

3.3.0.3 Muxing logic for choosing the tag and standard TAGE functionality

Here, we choose the prediction and tag from the tiles, for each table's output. The last 3 lines depict the familiar TAGE behavior where we have cascading muxes, from where we choose between a default base prediction, output of history table 0 and so on and so forth. It is shown in Figure 3.16 (page 16)

```
61          tiles[0].input_index = scatter_index[0];
62
63          if (scatter_index[0][2:1] == 0) begin
64              tiles[0].buff_sel = 1;
65          end else
66              tiles[0].buff_sel = 0;
67
68          tiles[1].input_index = scatter_index[1];
69
70          if (scatter_index[1][2:1] == 1) begin
71              tiles[1].buff_sel = 1;
72          end else
73              tiles[1].buff_sel = 0;
74
75          tiles[2].input_index = scatter_index[2];
76
77          if (scatter_index[2][2:1] == 2) begin
78              tiles[2].buff_sel = 1;
79          end else
80              tiles[2].buff_sel = 0;
81
82          tiles[3].input_index = scatter_index[3];
83
84          if (scatter_index[3][2:1] == 3) begin
85              tiles[3].buff_sel = 1;
86          end else
87              tiles[3].buff_sel = 0;
```

Figure 3.15: Input To Tiles

```
89  tile0_pred_out = (tiles[0].buff_sel)? ((scatter_index[0][0])? tiles[0].line1.prediction: tiles[0].line0.prediction): 'x;
90  tile1_pred_out = (tiles[1].buff_sel)? ((scatter_index[1][0])? tiles[1].line1.prediction: tiles[1].line0.prediction): 'x;
91  tile2_pred_out = (tiles[2].buff_sel)? ((scatter_index[2][0])? tiles[2].line1.prediction: tiles[2].line0.prediction): 'x;
92  tile3_pred_out = (tiles[3].buff_sel)? ((scatter_index[3][0])? tiles[3].line1.prediction: tiles[3].line0.prediction): 'x;
93
94  tile0_tag_out = (tiles[0].buff_sel)? ((scatter_index[0][0])? tiles[0].line1.tag: tiles[0].line0.tag): 'x;
95  tile1_tag_out = (tiles[1].buff_sel)? ((scatter_index[1][0])? tiles[1].line1.tag: tiles[1].line0.tag): 'x;
96  tile2_tag_out = (tiles[2].buff_sel)? ((scatter_index[2][0])? tiles[2].line1.tag: tiles[2].line0.tag): 'x;
97  tile3_tag_out = (tiles[3].buff_sel)? ((scatter_index[3][0])? tiles[3].line1.tag: tiles[3].line0.tag): 'x;
98
99  table0_pred_out = (history_vector[0])? tile1_pred_out: tile0_pred_out;
100 table0_tag_out = (history_vector[0])? tile1_tag_out: tile0_tag_out;
101 table1_pred_out = (history_vector[1])? tile3_pred_out: tile2_pred_out;
102 table1_tag_out = (history_vector[1])? tile3_tag_out: tile2_tag_out;
103
104 assign mux0 = (tag[0] == table0_tag_out)? table0_pred_out: base_prediction;
105 assign mux1 = (tag[1] == table1_tag_out)? table1_pred_out: mux0;
106 assign prediction = mux1;
```

Figure 3.16: RTL MUX logic

CHAPTER 4

RESULT ANALYSIS

4.1 *gem5* Results

Simulation of the *gem5* model was carried out without the usage of the Scoring Unit, as its logic is still a work in progress. But two simulations have been done to show how Dynamic TAGE can be effective.

4.1.1 Simulation 1

The configuration vector for the TAGE tables were all set to 4. Simulations results were obtained. They are mentioned in *stats_pre.txt* file in the ZIP folder. After the simulation, the table allocations were changed according to the number of access. The configuration vector was changed accordingly to ”allocated” more tiles to most accessed tables. Then the simulation two was carried out. The spectre program was simulated. The BTB misses were 5004 for Simulation 1.

4.1.2 Simulation 2

The modified configuration vector resulted in decreasing BTB misses to 4890. They are mentioned in *stats_post.txt* file in the ZIP folder.

CHAPTER 5
CONCLUSION AND FUTURE SCOPE OF WORK

5.1 Conclusion

From our exploration of dynamic TAGE through the gem5 simulation, we were able to demonstrate the efficacy of the dynamic TAGE implementation over standard L-TAGE and SC_L_TAGE methodologies.

5.2 Future Scope of Work

The results from the RTL and Gem5 simulations of the dynamically sizing the TAGE branch prediction have been encouraging, highlighting several directions for further research and development. Key areas for the future exploration include algorithm refinement for enhanced accuracy, reduced computational load and minimize power consumption. The potential integration of the machine learning techniques, offers an advanced machine learning method to further improve the performance of prediction. Finally, the goal is to move beyond simulations to test the predictor in real-world conditions, validating its effectiveness in practical situations and ensuring that it meets the security standards required for modern processors.

5.3 Contributions

- **Ayan Deep Hazra**: RTL Implementation in Verilog
- **Kiranmay Sekar**: gem5 implementation and CBP-16 Trace simulation
- **Pritheshwar Thirugnanasambantham**: gem5 implementation and benchmark simulations
- **Tingbiao Li**: Create the the format and most of the contents of ppt and report, the testbench of RTL, and the part of .py file in gem5

.

References

[1] D. J. Schlais and M. H. Lipasti, ”BADGR: A practical GHR implementation for TAGE branch predictors,” 2016 IEEE 34th International Conference on Computer Design (ICCD), Scottsdale, AZ, USA, 2016, pp. 536-543, doi: 10.1109/ICCD.2016.7753338

[2] André Seznec. Tage-sc-l branch predictors. In JILP-ChampionshipBranch Prediction, 2014

[3] Pruett, S., Zangeneh, S., Fakhrzadehgan, A., Lin, B., Patt, Y.N. (2016). Dynamically Sizing the TAGE Branch Predictor.

[4] M. Yoon, “Aging bloom filter with two active buffers for dynamic sets” in Knowledge and Data Engineering, IEEE transactions on, vol. 22, no. 1, pp. 134-138, 2010