POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Advanced Computer Architecture Final Project - Branch Prediction

Master of Science in High Performance Computing Engineering

Author: Alesio Demiri, Leonardo Evi

Advisor: Prof. Cristina Silvano

Co-advisor: Marco Ronzani

Academic year: 2024-2025

1. Introduction

Modern computer architectures heavily rely on instruction-level parallelism to achieve high performance. One of the most impactful techniques introduced is the pipeline, which allows multiple instructions to be overlapped, each one occupying a different unit of the pipeline. While pipelining significantly improves throughput, it also introduces new challenges. Among them, the problem of control hazards occurs when the processor encounters a branch instruction whose outcome is not yet known.

Branches (i.e. if statements and loops) determine the flow of execution. In a pipelined processor, when a branch instruction is fetched, its condition may not yet be resolved, and the processor cannot know whether to continue executing the next sequential instruction or jump to a different address. This could lead to pipeline stalls, where the processor must wait until the branch is resolved, delaying the execution and wasting resources.

To mitigate this, branch predictor were introduced. The main idea behind these components is to "guess" the outcome of a branch before it is resolved and speculatively execute instructions along the predicted path. If the prediction is correct, the pipeline continues without interruption. If it is incorrect, however, the speculatively executed instructions must be flushed, and the correct instruction must be retrieved, causing performance penalties. The accuracy of branch prediction is therefore critical to overall system performance. A highly accurate predictor can significantly reduce the number of pipeline flushes and stalls, allowing the processor to maintain high throughput.

Over time, branch predictors have evolved from simple static schemes (i.e. always predict not taken) to sophisticated dynamic predictors that learn patterns of branch behaviour across program execution. Among the most advanced of these is the TAGE (TAgged GEometric history length) predictor, which uses multiple prediction tables indexed with varying history lengths to capture both short and long term correlations in branch behaviour.

This paper presents our implementation of the TAGE branch predictor in C++, following the framework and test traces provided by the CBP competition committee (`https://ericrotenberg.wordpress.ncsu.edu/cbp2025/`). We discuss the predictor's architecture, specifying its update and prediction logic, and evaluate its performance using a set of benchmark traces.

2. High Level Structure

Our branch predictor is heavily inspired by the well known TAGE [1] branch predictor, that makes use of a single tagless table and multiple partially tagged history tables to perform a prediction.
In particular our predictor makes use of:

- A 260-bits global history register (GHR), that keeps track of the most recent conditional branches outcome (implemented as a shift register).
- 1 tagless table (i.e. the *base* table or the *base* predictor, T0) that performs predictions that do not include any information stored in the GHR.
- 8 partially tagged tables (T1, T2, ..., T8) to perform predictions that include information stored in the GHR. The idea is that each table will use more information stored in the GHR to perform its prediction.

Further details about the parameters can be found in the parameters.h file [2].
An entry in the base table is a simple 2-bits Finite State Machine that holds the information necessary to perform a prediction. Its behaviour is simple, the prediction for a branch is TAKEN until the prediction is incorrect for 2 times in a row, and vice-versa as shown in Figure 1.

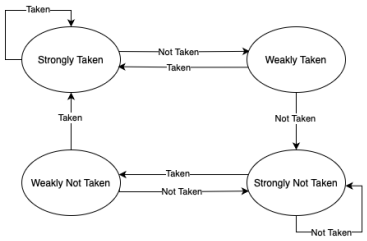

Figure 1: Transitions of a 2-bits FSM.

An entry in the partially tagged tables also includes:

- A 2-bits saturation counter (also referred to as u) that stores a value that can be interpreted as the "usefulness" of the table entry
- A variable size register that stores the tags. This field is necessary to (partially) avoid conflicts between different branches.

A simplified structure is shown in Figure 2.

3. Predict Policy

The prediction for a branch is purely based on the state of the branch predictor and the specific branch instruction Program Counter (PC). In this section we will see how, given the PC of a conditional branch instruction, it is possible to produce the corresponding prediction: TAKEN or NOT TAKEN.
First, we will discuss how each table can be accessed to get a prediction and then how the final prediction is computed. To get a prediction from the base table, we only need to map the program counter to a value that can be used as an index to access the table. In our case, we simply use the modulo operator, essentially considering only a few of the least significant bits of the PC to index the table. Then we just need to look at the state of the FSM to get the prediction.
The tagged table works similarly, however it must also include information stored in the GHR when performing a prediction. To do so, we first perform an HASH operation to combine the PC with a part of the GHR. It is important to say that each table will use a different number of (least significant) bits stored in the global history, with T1 using the minimum number and T8 the maximum. In our case, T1 to T8 will use, respectively, [5, 9, 15, 25, 44, 76, 130, 260] bits of global history. The choice of geometrically increasing sequence of global history lengths has been discussed in [1].
After computing the HASH, the modulo operator is applied to access the correct position in the table. At this point we must mention that a tagged table may not be able to produce a prediction; this happens when the tag field does not correspond to the remaining bits of the HASH. It is also important to say that each table will store only a portion of the remaining bits of the HASH and only that portion is considered during the comparison.
The final prediction is simply the prediction provided by the table that considered the longest part of the GHR when performing the HASH.

4. Update Policy

When a conditional branch instruction is executed, its PC and actual outcome (TAKEN or NOT TAKEN) are provided to the branch predictor and the component's state is updated accordingly. In this section we will describe such

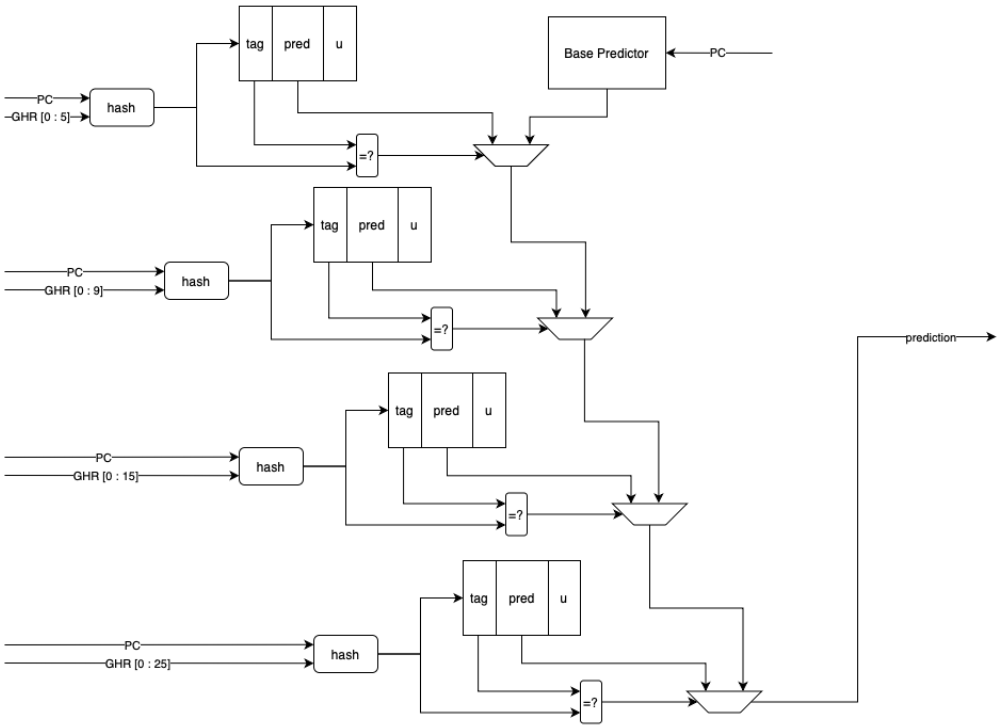

Figure 2: TAGE structure.

update policy.
First we should introduce some terminology that will be useful later. We will refer to the table that ultimately supplies the prediction as the "provider", and the prediction that would have been used if the provider had no matching tags as the "altpred".
The update process for TAGE involves the FSM-state of the predicting entry and u.

4.1. Updating the Usefulness Counter

The usefulness counter of the provider entry is updated only when the prediction from the provider and the alternate prediction (altpred) differ. In such cases, if the provider's prediction turns out to be correct, the u counter is incremented, reinforcing that the entry is beneficial. Conversely, if the prediction was wrong, the counter is decremented, reflecting reduced utility.
To prevent entries from being marked useful indefinitely, TAGE includes a mechanism for periodically aging the u counters. This is done gracefully in two alternating steps: in one pass, the most significant bit of each u counter is reset to zero; in the following pass, the least significant bit is reset. This aging step occurs every 256,000 branch instructions.

4.2. Updating on Correct Prediction

When a prediction is correct, only the FSM state of the provider entry is updated, according to the 2-bits FSM shown in Figure 1. No new entries are allocated, as the current predictor performed well.

4.3. Updating on Incorrect Prediction

In the event of a misprediction, the following actions are performed:

1. The FSM counter of the provider entry is updated first.
2. If the provider is not from the table with the longest history (i.e., it belongs to T_i where $i < M$), an attempt is made to allocate a new entry in a longer-history table T_k where $i < k < M$.

The decision to allocate a new entry is based on the usefulness counters of all longer-history components T_j (with $i < j < M$). The allocation follows these rules:

- (A) Allocation Criteria: If any of the examined entries has a usefulness counter $u = 0$, then allocation is permitted in the table with smaller index. Otherwise, if all candi-

date entries are considered useful ($u > 0$), no new entry is allocated. Instead, their u counters are decremented to gradually make room for future allocations.

- (B) Initialization: When an entry is allocated, its FSM state is initialized to the weakly taken state, the tag is assigned and the usefulness counter is set to 0, marking it initially as not useful until proven otherwise.

This careful update mechanism ensures that entries with longer histories are only added when needed, and only when space can be justified, which improves predictor stability.

5. Results and Benchmarks

To evaluate the effectiveness of our TAGE branch predictor implementation, we used the trace-based evaluation infrastructure provided by the Championship Branch Prediction (CBP) framework. This framework allows for consistent benchmarking across different branch prediction strategies by using a standard set of program traces that simulate realistic workloads.
The CBP Committee offered traces from various applications domains labeled as: int, fp, infra, compress and web.
The primary metric we considered is the misprediction rate, expressed as a percentage of incorrectly predicted branches over the total number of branches. Lower values indicate better performance.
The table below summarizes the misprediction rate of our TAGE predictor across several trace categories:

TAGE Misprediction Rate (%)

	INT	FP	INF	WEB	CMP
MR	11.9%	10.8%	9.6%	11.3%	6.1%

Table 1: Misprediction rates of the TAGE predictor across different CBP benchmark categories.

As shown, our predictor achieved consistent performance across diverse domains, demonstrating the generalization capability of the TAGE algorithm. This results depend on history lengths, table sizes, and tuning parameters.

6. Conclusions

In this project, we successfully implemented a TAGE branch predictor in C++, following the specifications and evaluation framework provided by the Championship Branch Prediction (CBP) competition. Through this work, we explored the internal mechanics of one of the most accurate state-of-the-art predictors, focusing on the role of tagged components, prediction counters, and usefulness-based entry management.
Our implementation [2] demonstrated consistent prediction accuracy across a range of benchmark categories, highlighting the robustness of the TAGE architecture.

References

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

[2] Alesio Demiri and Leonardo Evi. `https://github.com/alesiodemiri/ACA-BP`, 2025.