IMPERIAL

MEng Individual Project

Imperial College London

Department of Computing

Augmenting Dynamic Branch Predictors with Static Transformer Guided Clustering

Author:
Avaneesh Deleep

Supervisor:
Prof. Paul Kelly

Second Marker:
Prof. Peter Pietzuch

June 16, 2024

Abstract

Branch prediction is a fundamental aspect of modern high-performance processors. However, despite its critical importance, significant breakthroughs in this field are few and far between. Traditional predictors have primarily relied on recent branch history, overlooking other potential sources of contextual information.

With the recent attention garnered by Natural Language Processing, we have seen some research attempting to build new branch predictors that use Transformer Neural Network models. However, these models' size, long training time, and nature require that learning is done at compile-time and never changes during run time.

This thesis introduces a novel approach to branch prediction by integrating the compile-time analysis of a transformer-based model into a state-of-the-art dynamic predictor mechanism. Key findings of our research include:

- The integration of information derived from a transformer model improves branch prediction accuracy, outperforming traditional static and state-of-the-art dynamic predictors.
- Our custom branch predictor, when fed with the colour labels generated from our transformer-based and takenness-based clustering algorithms, produces an accuracy improvement of 3.0% over the state-of-the-art TAGE-SC-L 64KB branch predictor while maintaining a smaller memory footprint. Furthermore, we see up to 4.88% improvement in Misses-per-Kilo-Instruction in some programs compared to the current state-of-the-art Tournament Predictor.
- Our custom predictor architecture demonstrates robustness and efficiency in handling complex program flows, showcasing its potential for real-world applications. Its flexible design opens up the predictor to learn from context never before used in branch prediction.
- This research presents a novel perspective on branch prediction methodologies, highlighting the importance of static analysis in complementing dynamic predictors for more efficient CPU designs.

This thesis contributes to the evolution of branch prediction techniques, opening up a new compile-time machine learning analysis can enhance prediction accuracy and overall CPU performance in modern computing environments.

Acknowledgements

I extend my sincere thanks to Professor Paul Kelly for his invaluable guidance, Luke Panayi for his assistance, and my family for their unwavering support. I am also grateful to my friends for their encouragement throughout the research process for this thesis.

Contents

Chapter 1

Introduction

In the realm of modern computing, the efficient execution of program instructions is paramount, and at the heart of this efficiency lies the branch predictor - a critical component in the CPU that anticipates the outcome of conditional branches. As the complexity of software continues to grow, the need for accurate and adaptive branch prediction becomes increasingly pronounced.

In contemporary processors, Branch Prediction is a crucial technique, seeking to forecast the outcomes of program branches before their actual execution concludes. This strategic approach serves to mitigate potential delays waiting for branch conditions to resolve, allowing for the proactive fetching of subsequent instructions. The accuracy of the branch predictor emerges as a pivotal factor, significantly shaping the overall performance of these designs. Any incorrect branch predictions have a profound impact on processor efficiency, and minimising these mistakes is an integral way to maximise compute performance in a post-Moore's Law world.

However, modern branch prediction methods that make predictions based on dynamic recent branch history, all approach a theoretical limit bounded by the finite information they access and their compression of that information [1]. While this limit is high, further significant improvements in branch predictor performance are now only possible by consuming more information, or context, than previous designs. To break past this information barrier faced by the state-of-the-art predictors we must rethink how a branch predictor learns, what it learns from, and find the best ways to compress new context into better predictions.

Moreover, the advancement of machine learning has opened avenues for enhancing branch prediction strategies. This thesis explores the integration of a transformer-based approach at compile time to analyse code patterns and label instructions in the compiled code with "colours" according to their behaviour. Clustering branches with colours offers a novel source of additional information that can be used to augment dynamic branch predictors. The static processing at compile time influences dynamic prediction through a customised branch predictor, thereby optimising prediction accuracy and ultimately improving processor speed. It preserves the inherent accuracy advantage of dynamic branch prediction by leveraging the runtime program state while incorporating the advantages of static prediction such as larger model sizes and longer prediction times.

1.1 Objectives

This research project endeavours to delve into the realm of branch prediction methodologies, seeking innovative ways to enhance prediction accuracy and, consequently, overall CPU performance.

The primary objective of this thesis is to advance the current state of branch predictors and expand the information sources branch predictors learn from by leveraging a transformer-based approach. The project aims to develop a branch predictor that learns dynamically but, utilises static information in the form of colour labels on branches to improve prediction. These colours can be derived from a transformer trained on traces from the program to cluster branches into families.

A key problem with modern branch predictors is pollution. As a predictor is learning how one particular branch behaves, the execution of a completely separate branch behaviour pattern can pollute and undo all the learning relevant to the first branch. The pollution causes the predictor to make mistakes in both the first and second-branch behaviour patterns. We aim to reduce pollution by the use of the extra context provided by colour labels. If we can identify branches

that may exhibit destructive interference at compile time and ensure they are assigned different colour labels, we can augment the branch predictor at runtime to be selective in the branch history it uses and make predictions based only on branches of the same colour. This enhancement supplies the branch predictor with extra information it previously did not know, mitigates pollution effects and enhances the accuracy of branch prediction. An improvement in branch prediction accuracy would improve cutting-edge CPU performance if implemented efficiently in hardware.

To fulfill this goal we have two key components. Firstly, the hardware design component. We aim to design and simulate a fully functional novel branch predictor that, at run time, can make use of colour labels that supply extra context about the nature of the branch instructions. Secondly, the compile time branch clustering algorithm component. We aim to experiment with multiple ways to cluster branches according to their nature and separate branches that may interfere destructively with different colour labels. We will experiment with statistical methods and transformer-based methods for clustering branches into colours.

1.2 Contributions

In this work, we:

- **Generate useful colours for branches in the SPEC2006 benchmark suite at compile time:** We use the SPEC2006 CPU benchmark suite [2] as an indicator of the branch predictors utility in real-world computation workloads. We assign colours, also referred to as clusters, to instructions in the benchmark programs to add an extra layer of context the branch predictor can use to make better predictions. The type and source of context we inject are determined by how we generate the colour clusters. We generate clusters in two key ways. The first, explained in Chapter 5, is simply looking at the takenness of a branch in the training set. The second, explained in Chapters 6 and 7, is by extending previous work at purely static transformer branch prediction [3], with a unique approach to cluster branches by the transformer's learnt attention weights and changes in the transformer architecture to generate more useful attention matrices.

- **Design a custom branch predictor architecture:** We design and implement a novel branch predictor in the gem5 [4][5][6] microarchitecture simulator with a mechanism for it to read the branch colours that were generated by the transformer at compile time. Our novel branch predictor can then make predictions based on the dynamic history, like current state-of-the-art predictors, but also learn from a branch's colour label which carries extra context information. The design and motivation are described in Chapter 4.

- **Highlight the utility of adding colour information to instructions:** We test and evaluate several strategies for assigning colours to branches and branch predictor setups to show that the new augmented branch predictor outperforms the TAGE-SC-L 64KB, a purely static transformer approach [3], and the Tournament predictor. We see up to 4.88% improvement in Misses-per-Kilo-Instruction in some programs compared to a state-of-the-art Tournament Predictor. Our custom branch predictor can do this while still having a memory footprint smaller than the TAGE-SC-L 64KB.

- **Provide mathematical proof on the bounds of utility for our local colouring scheme:** We prove our colouring methodology for a local branch predictor can always provide statistically better predictions while maintaining an identical memory footprint to the state-of-the-art. We provide the assumptions and bounds for this claim to be true, and why it holds for most programs. To avoid distracting from the experimental nature of the remainder of this thesis these proofs are detailed in Appendix A.

- **Opens up new directions in which future research into branch prediction can take:** By creating a branch predictor independent of the colouring scheme, we provide the framework for future work to colour branches in unique and powerful ways. New ways of colouring branches can inject any number of contexts and data sources into our novel branch predictor. We have created a flexible predictor architecture that can learn from information never before used in branch prediction and opens up a new way of looking at improving branch prediction. We suggest some ideas and explanations for new colouring methodologies that may be fruitful in producing even better prediction results.

1.3 Overview of our Proposed New Branch Prediction Method

The entire process we develop for improved branch prediction of a program is summarised at a high level here: (The specifics of each step are revisited and explained thoroughly in their appropriate chapters)

1. Generate traces of the program:
    1.1. Use BBV to profile the program and generate basic block vectors for it.
    1.2. Pass the BBVs to SimPoint to produce SimPoints.
    1.3. Use the SimPoints to generate checkpoints with Gem5.
    1.4. Use the checkpoints to produce full traces with Gem5.
    1.5. Extract key information such as branch addresses and taken direction from traces.
2. Use the trace statistics to produce local colour labels for the branches we have seen. We develop a takenness-based method to produce these colour labels:
    2.1. Branches in the traces that are taken more often than not are assigned a different colour label to branches that are taken less often. One cluster is assigned colour 0 and the other colour 1.
    2.2. At runtime branches unseen in the traces are given the local colour label 0 which puts them in the same cluster as other branches seen in the traces.
3. Use the trace statistics to produce global colour labels for the branches we have seen. We develop a rolling window transformer-based method to produce these colour labels:
    3.1. Train our transformer model to predict branch direction based on the traces of the program.
    3.2. Harvest the learnt attention from the transformer as it provides insight on what branches are important to the outcome of others.
    3.3. Treat the attention between branches like a graph and use fast greedy community detection to partition the branches into clusters. Assign each cluster a different colour.
    3.4. At runtime branches unseen in the traces are given the global colour label 0 which is reserved for these unseen branches.
4. Feed the local and global colour labels to our novel branch predictor which is an augmented tournament predictor so it has independent local and global predictors. The local and global predictors get extra context from the local and global colour labels respectively which helps reduce destructive aliasing.

1.4 Real World Application

If our novel branch predictor and clustering algorithms for generating colour labels were implemented in real-world silicon, we anticipate significant improvements in branch predictor performance. Users of CPUs equipped with our branch predictor can opt to increase compile-time work to enhance runtime performance by generating traces and utilising clustering algorithms to produce colour labels. Alternatively, they can forgo colour labels, maintaining standard compile-time work, as our novel branch predictor can function identically to state-of-the-art predictors without them.

We believe users will choose to optimise with our novel branch predictor in scenarios where compile time is less critical than runtime performance. For instance, high-frequency trading applications, which prioritise minimising latency, would benefit from our branch predictor and colour labels, as the additional compile time is justified by the lower latency achieved. Conversely, in contexts where optimisation is unnecessary, such as during debugging, the colour labels can be omitted, similar to other compiler optimisations.

Ultimately, this research contributes to the current body of knowledge by introducing a novel perspective on branch prediction. The utilisation of a transformer-based approach and the incorporation of clustering techniques offer a unique angle for enhancing the adaptability and precision of branch predictors. The project's findings have the potential to influence the design and implementation of future CPUs, offering more efficient solutions for handling complex program flows.

Chapter 2

Background and Related Work

In this chapter, we will outline the key concepts of branch prediction and previous research on surrounding topics to set the thesis into context.

2.1 Preliminary Knowledge

This section aims to clarify key ideas driving all branch predictions and introduce the relevant architectural units. This section will introduce major concepts and terminology that will be used throughout the remainder of this thesis.

2.1.1 Branch Instructions

Branch instructions are an integral part of controlling the flow of a program and are present in every Instruction Set Architecture (ISA). A branch instructions' purpose is to inform the program to jump to a different location in the code. As such, they are required when any programming language calls for if-statements, loops, or other conditional statements. Branch instructions have two key properties that determine how they behave. The first is whether the branch is conditional or unconditional, and the second is whether the branch target is encoded directly or indirectly.

Conditional and Unconditional Branches

Conditional branches require run-time evaluation of branch conditions, allowing the program to follow different paths on every occurrence of the branch. On the other hand, Unconditional branches enforce an unequivocal shift in program flow, independent of conditions.

Conditional branches, exemplified by constructs such as the *if-else* statement, cause jumps in the Program Counter (PC) based on specified conditions.

```c
void conditional_jump(int x) {
    if (x > 5) {
        printf("Condition is true: x is greater than 5\n");
    }
}
```

Listing 2.1: Conditional branch example in C

```assembly
conditional_jump:
    cmp r0, #5          // Compare x with 5
    ble end_function    // Branch to end_function if x is less than or equal to 5
    // Code for true condition
    ldr r0, =format_true
    bl printf

end_function:
    bx lr
```

Listing 2.2: Conditional branch example in ARM Assembly

Unconditional branches dictate an immediate shift in program flow, irrespective of conditions. This is often seen as the ARM Assembly's unconditional branch statement, *b*. In the context of branch

prediction, we are unconcerned with unconditional branches as their program path is fixed and deterministic at every run.

```
example_function:
    // function code here

    b exit_function // Unconditional jump out of function

exit_function:
    bx lr
```

Listing 2.3: Unconditional branch example in ARM Assembly

Direct and Indirect Branches

Branches, both unconditional and conditional, need to specify a target for where they want to program counter to jump to. Direct branches specify the target inside the instruction itself while indirect branches do not specify the destination address explicitly in the branch instruction. Instead, the instruction refers to a memory location or register that contains the target address. The actual target address is determined at run-time.

A common use of the indirect branch is the *bx lr* instruction, which is used to branch back to the link register *lr*, effectively returning from the subroutine.

```
_start:
    b target_address // Unconditional direct branch to 'target_address'

    // Code for the remainder of the function ...

target_address:
    // Code at the target address
    // ...
    bx lr // Unconditional indirect branch back to the link register 'lr'
```

Listing 2.4: Unconditional direct and indirect branch examples in ARM Assembly

2.1.2 Purpose of Branch Prediction

Before delving into the intricacies of current branch predictors, we must first consider why branch prediction is a useful method to speed up program execution in modern processors.

A processor operates on a sequence of instructions presented in machine code, and conventionally, it executes these instructions in order. However, the execution of a single machine code instruction is not a straightforward, atomic operation. It involves distinct stages within the processor's pipeline. A typical execution process unfolds as follows:

- **Instruction Fetch (IF):**
  - Fetches the next instruction from memory based on the program counter (PC).
  - Increments the PC to point to the next instruction.
- **Instruction Decode (ID):**
  - Decodes the fetched instruction to determine the operation to be performed.
  - Identifies the operands and their addresses.
  - **Branch Prediction:** Predicts the outcome of conditional branches based on historical behaviour or patterns.
- **Execution (EX):**
  - Executes the operation specified by the decoded instruction.
  - Arithmetic and logic operations are performed in this stage.
- **Memory Access (MEM):**
  - If necessary, accesses memory to read or write data.
  - Data cache or main memory may be accessed depending on the design.

- **Write Back (WB):**
  - Writes the results of the executed instruction back to registers.
  - Updates register values with the result of the computation.

There can be a differing number of stages depending on processor architecture, and they are commonly referred to as pipeline stages.

Modern CPUs are pipelined such that multiple instructions are "in-flight" at different stages of the pipeline simultaneously. This allows the processor to work on different stages of various instructions concurrently, enhancing throughput. In an in-order execution model, earlier instructions must always be ahead of later ones in the pipeline and these stages progress sequentially for each instruction. This staggered but sequential processing can be seen in Figure 2.1.

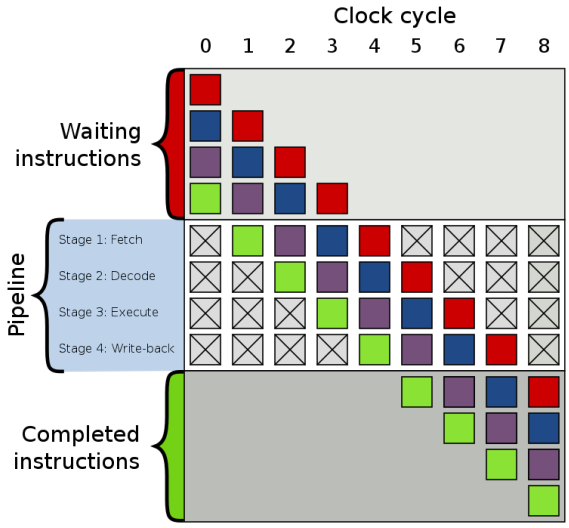


Figure 2.1: Example of a 4-stage pipeline. The coloured boxes represent instructions independent of each other [7].

However, this linear progression can lead to inefficiencies, especially when dealing with conditional branches. For example, the processor often encounters situations where the outcome of a branch instruction depends on the execution of preceding instructions. These preceding instructions are still "in-flight" ahead of the branch in the pipeline and their outcome has not yet been resolved. The processor would have no choice but to wait for the determination of this outcome, which forces it to execute only one instruction at a time, lose the throughput benefit of pipelining, and introduce pipeline stalls, causing idle time in the processor.

This is where branch prediction becomes crucial. Branch prediction anticipates the likely outcome of a branch instruction before it is officially resolved. By making educated guesses based on historical behaviour and runtime information, branch prediction enables the processor to speculatively execute instructions along the predicted branch path. This speculative execution helps in mitigating the impact of pipeline stalls, enhancing parallelism, and ultimately contributing to improved program execution speed. However, it is important to note that incorrect predictions can lead to steep performance penalties, as all false speculative execution needs to be undone.

Cache latencies mean instructions often don't have the required data to operate on (operands) available immediately. This latency, along with limited arithmetic units, leads to the desirability of out-of-order execution, allowing the CPU to execute instructions as soon as their operands are available, rather than waiting for earlier instructions to complete. This approach increases processor throughput and efficiency. In an out-of-order execution model, instructions can be dynamically reordered and executed as long as their data dependencies are satisfied. This flexibility maximises the utilisation of execution units, cache, and other resources within the CPU.

Most modern CPUs adopt out-of-order execution to leverage these advantages, which also introduces challenges related to branch prediction. When encountering a branch with undetermined parameters, CPUs may stall until the correct branch path is determined. Incorrect branch predictions incur a significant performance penalty, as all erroneously executed instructions must

be discarded or 'squashed'. This penalty underscores the critical importance of accurate branch prediction mechanisms, an area of intensive research over the past decade.

2.1.3 Branch Predictor Methodologies

The ultimate goal of static branch prediction is to perfectly predict what path a program will take before executing any instructions at all. However, this is considered impossible because of the inherent uncertainty in program flow. Conditional branches introduce a level of unpredictability as their outcomes are often not known until execution.

On the other hand, dynamic branch predictors operate by leveraging historical information, specifically the history of branches. They track the recent outcomes (taken or not taken) of individual branch instructions, storing this data in a Branch History Table or a similar structure. The predictor analyses this historical context to make real-time predictions about the future behaviour of branches during program execution.

The key insight lies in the idea that the more information we gather, whether through dynamically observing the program run or statically analysing pre-runtime metrics, the better the predictor becomes. A more comprehensive view of the program's behaviour allows for more effective compression of predictive information. Just as compressing data involves representing it more efficiently, optimising branch prediction involves distilling complex program behaviours into predictive patterns.

Whether dynamically monitoring execution patterns or statically analysing the program's structure, the aim is to extract meaningful insights that enhance the predictor's accuracy. This approach aligns with the notion that the more we understand a program's behaviour, the better we are at predicting its future paths.

2.1.4 Basic Dynamic Prediction Methods

Key Hardware Elements for Dynamic Branch Prediction

The Program Counter (PC) is a register that holds the address of the current instruction being executed in a processor, and it is crucial for indexing branch prediction structures because it helps identify and distinguish different points in the program, enabling accurate predictions based on the historical behaviour of branches at specific locations.

The Branch History Register (BHR) is implemented as a shift register and it is updated with every branch decision to store the historical outcomes of recent branch instructions in a processor. It is useful for indexing branch prediction structures as it captures the sequential behaviour of branches, aiding in predicting future outcomes based on past patterns.

Bimodal Predictor

The Bimodal Predictor, as seen in Figure 2.2 is one of the simplest dynamic predictors. It is a saturating counter, with the takenness history of a branch shifting the counter through states.

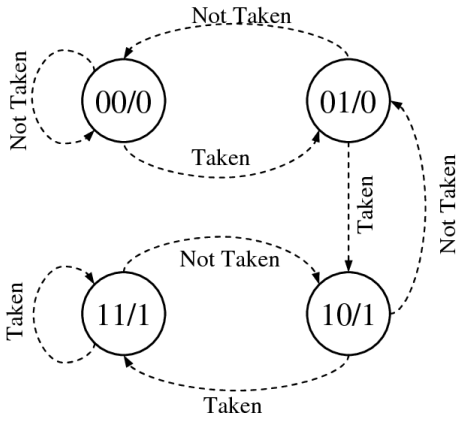


Figure 2.2: A 2-bit Bimodal Predictor finite state machine [8].

Branch History Tables

The Branch History Table (BHT), also referred to as the Correlation-Based Branch Predictor or Pattern History Table, is an array of Bimodal Predictors indexed by several lower bits of the Program Counter as seen in Figure 2.3. Multiple Bimodal Predictors are advantageous over a single one because they allow the system to maintain different prediction models for various branches based on their unique program counter values. This approach improves accuracy by tailoring predictions to the diverse behaviours of different branches, optimizing the ability to capture and adapt to varying patterns in the program's execution.

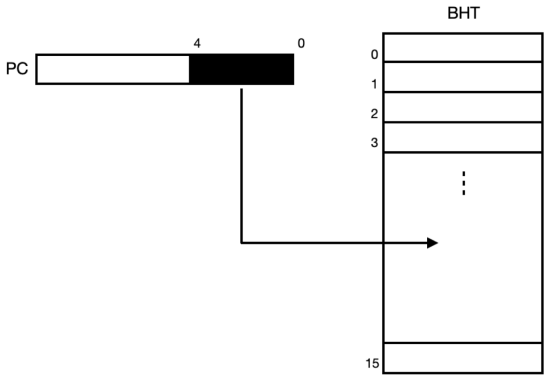


Figure 2.3: Branch History Table indexed with the 4 least significant PC bits. Every entry in the branch history table represents a 2-bit saturating counter [3].

Two Level Predictors

A Two-Level Predictor, as opposed to a simple Branch History Table (BHT), incorporates an additional level of complexity by using multiple tables. A two-level adaptive predictor excels in scenarios where branch outcomes depend on patterns of past occurrences. For instance, if an 'if statement' is executed four times, the decision on the fourth execution could rely on whether the previous three were taken or not. In such cases, a two-level adaptive predictor is far more accurate than a BHT or Bimodal Predictor alone. A Two-Level Predictor uses the least significant n bits of the BHR to remember the history of the last n occurrences of the branch and employs one saturating counter for each of the possible 2^n history patterns. This approach improves prediction accuracy for conditional jumps with regularly recurring patterns, providing a more nuanced and adaptable prediction mechanism.

The GSelect Predictor is an example of a Two-level Predictor as seen in Figure 2.4. It uses a series of BHTs, which are indexed with the Branch History Register. Then within the selected BHT, it indexes with the Program Counter.

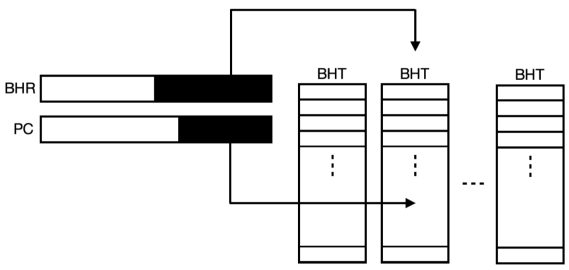


Figure 2.4: GSelect Predictor [3].

2.1.5 TAGE

The TAGE (Tagged Geometric History Length) predictor has been one of the most performant branch predictors since its proposal [9].

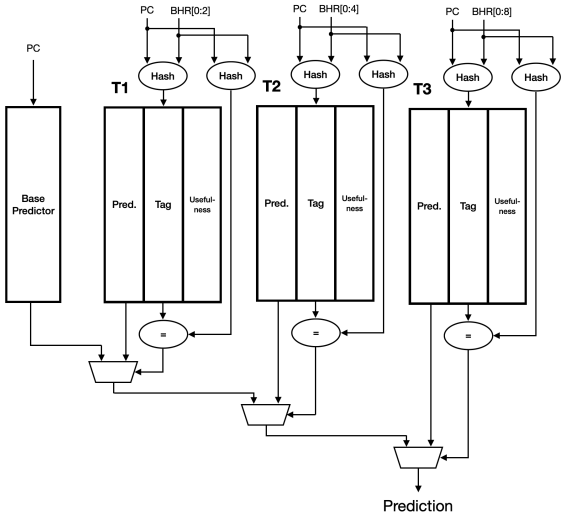


Figure 2.5: TAGE Predictor inner workings [3].

While the inner workings of the TAGE predictor 2.5 are not directly relevant to our research direction, it is crucial to understand how TAGE and other modern predictors learn and make predictions. TAGE operates by remembering the outcomes of branch instructions in a historical dataset. It combines a standard predictor and custom predictors. The standard predictor, like a finite state machine, records basic information about branch outcomes. On the other hand, custom predictors store more complex historical data, considering different lengths of past outcomes in a geometric series. This allows TAGE to adapt to diverse program patterns effectively. When a new branch instruction is encountered, TAGE looks back into its historical data, finding the best match to predict the current branch's outcome. This blend of standard and custom predictors enables TAGE to make accurate predictions by leveraging a rich history of branch behaviours.

2.1.6 Tournament

Different branch prediction schemes have different merits. They have varying prediction performance across different branch pattern scenarios and vary in time to train. Tournament predictors strive to merge multiple prediction methods into a unified structure, selecting the strongest method of the schemes available at every prediction call.

Tournament predictors train and run all the predictors they manage in parallel, and use a separate metric to select the optimal one for the situation. TAGE-SC-L [10], a tournament extension of the TAGE branch predictor, employs a dynamic global history length selection mechanism facilitated by a tournament predictor. It maintains multiple prediction tables with varying history lengths. The tournament predictor dynamically evaluates the recent accuracy of different history lengths and selects the most effective one for a given branch. This adaptive approach continuously refines its predictions, adjusting the influence of history lengths based on their performance, resulting in a branch predictor finely tuned to the evolving patterns in program execution at the cost of extra hardware dedicated to branch prediction.

The tournament predictor structure was proposed by Scott McFarling [11] with a duelling local and global predictor. McFarling describes several configurations of the tournament, but we will focus on the implementation used in the Gem5 simulator [4][5][6]. The local predictor seen in Figure 2.6, utilises a history table and counter array to make predictions based solely on the recent history of the current branch. Conversely, the global predictor in Figure 2.7 employs a single shift register per process, recording the directions taken by recent conditional branches, thereby offering a broader perspective by considering the global branch history.

In the tournament predictor, the choice between the predictions of the local and global predictors is determined by a selector or choice predictor as seen in Figure 2.8. This selector examines the past performance of both predictors and chooses the prediction deemed most reliable. Specifically, if the global predictor has consistently outperformed the local predictor over a specified period, the selector favours the global prediction. However, if the local predictor has demonstrated superior accuracy, its prediction is selected. This dynamic selection mechanism ensures that the

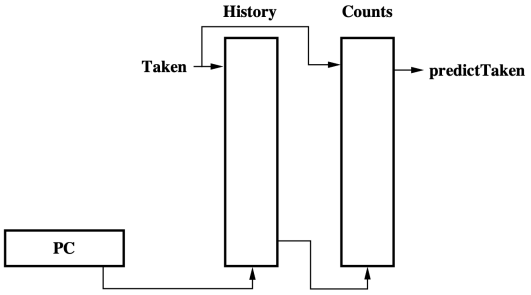


Figure 2.6: Local History Predictor Structure [11].

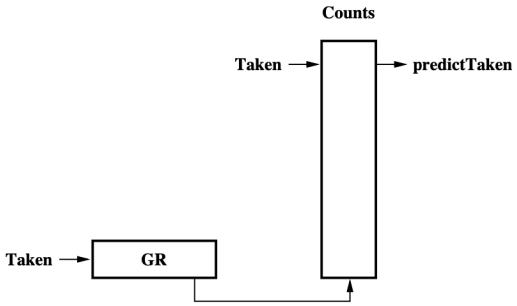


Figure 2.7: Global History Predictor Structure [11].

tournament predictor adapts to varying patterns and optimally utilises the strengths of both the local and global predictors. The global predictor's larger size and complexity, often mean it is more accurate than the local predictor but requires a long 'warm-up' period. The choice predictor works by checking the resolution of recently predicted branches. If the global predictor and local predictor predict differently, then the correct one gains a count for that branch's address. Once the count for the global predictor reaches a threshold it is deemed good enough to switch to.

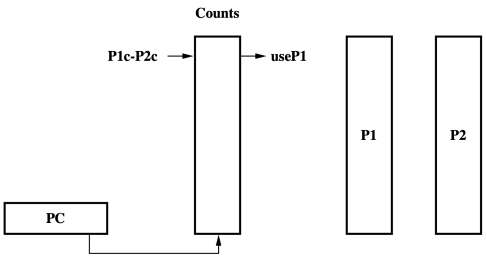


Figure 2.8: Choice Predictor Structure [11].

However, both the local and global predictors are limited in capacity. They only have a finite number of bits to track historical patterns, thereby indexing the counts table. Consequently, there is a risk of multiple branches hashing to the same entry in the history table, leading to potential conflicts and degraded prediction accuracy.

For the local predictor, the finite length of the history entry restricts its ability to capture complex patterns in branch behaviour. With a limited number of bits, the history table may become oversaturated, causing different branches to share the same history entry. This aliasing can result in contention among branches, where their behaviour patterns clash, complicating prediction and reducing accuracy.

Similarly, the global predictor faces challenges due to its reliance on a single shift register to record branch history. While the shift register provides a broader perspective by considering

the global history of conditional branches, its finite size imposes constraints on the granularity of historical information that can be captured. As a result, branches with distinct behaviour patterns may converge in the global history register, leading to ambiguity and potential clashes in prediction.

2.2 Related Work

This section aims to explore past research into surrounding areas. We will comment on the original authors' findings and how this informs our own research trajectory. The prediction schemes considered here are loosely informed by an overview by Intel [12] outlining modern state-of-the-art branch predictors.

2.2.1 Static Branch Prediction with Machine Learning

Static branch prediction methods produce a predefined branch prediction pattern before runtime. This can be as simple as the always-taken predictor, where conditional branches are always taken. Static predictors do not learn as they never see the true outcome for the branches they predict, so they perform significantly worse than dynamic predictors than improve at runtime. However, research by Calder et al. produced a more nuanced static branch predictor that is trained on seen programs to predict control flow in unseen programs [13]. Calder et al. call this approach to branch prediction 'evidence-based static prediction' (ESP). ESP trains a machine learning model on a corpus of programs, and the model is then used to infer the behaviour of new programs. In particular, they use neural networks and decision trees to map static features associated with each branch to a prediction that the branch will be taken. While competitive at the time of publication, the ESP method is not as performant as modern dynamic approaches, with an average miss rate of 20%.

A large limitation of the static predictor is its over-generalisation. Training a predictor to predict all branches across all programs, requires an incredibly complex model that can capture and identify all possible branch behaviour. However, a smaller dynamic predictor that is trained at runtime only needs to capture the local behaviour of that particular program it is predicting. The static predictor's large training set and generalisation provide it with more context about a branch, but, unlike a dynamic predictor, it has no information about the local history of the branch it is trying to predict. This local history is far more important in determining the 'takenness' of a branch than a wider context of branching across a corpus of programs.

2.2.2 Dynamic Machine Learning and Perceptron-based predictors

While the ESP static machine learning predictors [13] are trained before runtime, a dynamic machine learning strategy is trained at run time. This typically means a much smaller machine-learning model that is lightweight and simple enough to train and predict at runtime. We used work by Joseph [14] to guide our understanding of deep learning and machine learning dynamic branch prediction methodologies.

Jiménezg and Lin present a new method for branch prediction [15]. They observed that all existing two-level techniques use tables of saturating counters and attempt to improve accuracy by replacing these counters with neural networks.

A traditional neural network is composed of interconnected neurons. Each neuron, as depicted in Figure 2.9, is a product of a weight matrix and input signals from the previous layer in the network. The output signal then passes through an activation function that allows the neuron to capture complex behaviour better.

Due to the complexity of traditional neural networks, Jiménezg and Lin concluded they are prohibitively expensive to implement as branch predictors. Instead, they explore the use of perceptrons, one of the simplest possible neural networks. As seen in Figure 2.10, Perceptrons are less complex as they drop the activation function after every neuron. This makes them easier to implement in hardware.

Jiménezg and Lin built a two-level predictor but replaced each saturating counter in the BHTs with a perceptron. While saturating counters change their output based on their most significant bit of branch history, the perceptrons take the branch history as input and output a real value. The output real is mapped to 1 or -1 for taken or not taken predictions respectively.

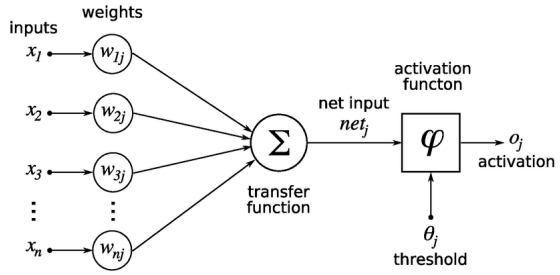


Figure 2.9: Neuron from traditional Neural Network [16].

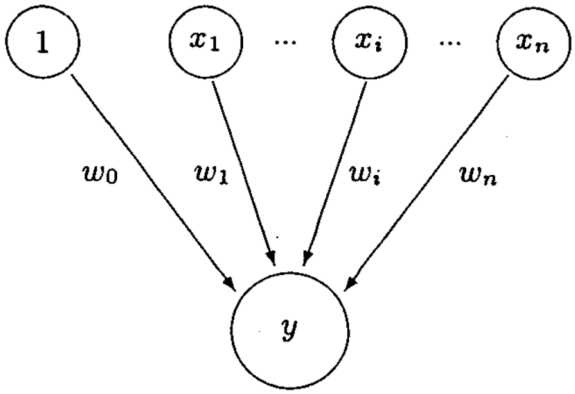


Figure 2.10: Perceptron as detailed by Jiménezg and Lin [15].

Upon learning the true outcome of a branch, the predictor performs one update step seen in Figure 2.11. The update changes the current weight by the error in the prediction of the prediction is significantly wrong in magnitude or completely wrong in prediction outcome.

```
if sign(y_out) ≠ t or |y_out| ≤ θ then
        for i := 0 to n do
                w_i := w_i + t x_i
        end for
end if
```

Figure 2.11: Perceptron update rule as detailed by Jiménezg and Lin [15]

Finally, the researchers assess the performance of their perceptron predictor using the SPEC2000 benchmark suite. The benchmark is commonly used in the industry and the more recent SPEC2006 suite will be used for our research. In this investigation, the behaviour of branches is categorised as either linearly separable or linearly inseparable. Linearly separable branches refer to a class of branches for which the decision boundary between taken and not taken outcomes can be effectively represented by a linear function such as the one outputted by a perceptron. The perceptron predictor demonstrated superior performance against a Gshare predictor with linearly separable branches, but Gshare performed better when branches could not be linearly separated. The choice of a perceptron as a predictor is motivated by its efficient hardware implementation. Although other neural architectures like ADALINE and Hebb were considered in this study, their performance was compromised by low hardware efficiency and accuracy.

After the paper's publication, Akkary et al. introduced a branch confidence estimator based on perceptrons to minimise branch mispredictions [17]. Mispredicted executions can have adverse effects on the system, utilising resources, causing execution stalls, and impacting power consumption due to the execution of additional instructions during prediction misses. In deeper pipeline processors, pipelining gating becomes crucial for minimising wasted executions resulting from incorrect speculative decisions made by predictors. By implementing pipeline gating, the processor can temporarily halt the progression of instructions in the pipeline, preventing the execution of further speculative instructions based on the current incorrect predictions. The perceptron-based

branch confidence estimator by Akkary et al. offers multi-valued outputs instead of a binary taken or not taken output as seen in Figure 2.12. The predictor classifies branch instructions such as "strongly low confident" and "weakly low confident", which allows more efficient pipeline gating when multiple low confidence branches are speculatively executed. The perception network functions similarly to Jiménezg and Lin's model, but instead of only outputting 1 or -1 from the real output of the perception, the output is compared against a specific threshold λ. When the output difference is larger than the threshold, the prediction has low confidence, but if the output is smaller than the threshold then the prediction is classified as having higher confidence.

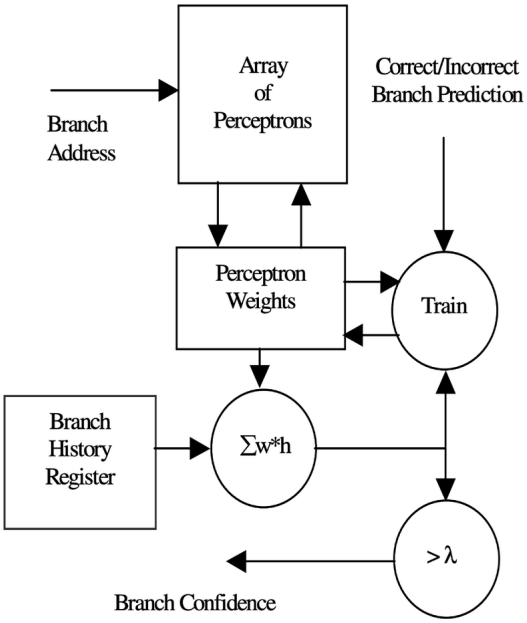


Figure 2.12: Perceptron confidence estimator [17].

Further research into perception dynamic branch predictors has followed both Jiménezg and Lin and Akkary et al.'s papers, to improve the power of the model incrementally. A perceptron-based approach can be seen in commercial AMD x86 CPUs in their Ryzen architecture [18]. While AMD has not made the specifics of their proprietary branch predictor publicly available, it is clear the dynamic machine learning approach has plenty of merit.

2.2.3 Using profiling to Blend Static and Dynamic Predictors

Modern branch predictors are largely dynamic as they perform significantly better than their static counterparts that cannot learn from program behaviour. Unfortunately, the majority of dynamic predictors encounter the 'aliasing problem' when two branches occupy the same position in the dynamic predictor. When these two branches exhibit distinct behaviour, such sharing can lead to repetitive mispredictions by the dynamic predictor, a phenomenon known as destructive aliasing. Conversely, when sharing enhances prediction accuracy, it is termed constructive aliasing.

Patil and Emer [19], try to address this aliasing problem by using profile-guided analysis to identify branches that are predicted poorly dynamically and separating these to be predicted statically at run time.

The authors identified which branches to predict statically in two phases. Initially, during the selection phase, they made decisions about which branches in their test programs would be statically predicted and defined their static predictions. In the subsequent phase, they executed the simulation of a dynamic predictor using static hints obtained earlier. The focus was on statically predicting two types of branches: those deemed easy for the dynamic predictor to predict and those considered difficult. The rationale behind this approach was to optimise the utilisation of resources in dynamic predictor tables. The authors hypothesised that branches falling into the first category (easy to predict dynamically) allocate dynamic resources inefficiently. On the other hand, branches in the second category were anticipated to benefit more from static prediction due to challenges faced by the dynamic predictor in accurately predicting them.

To identify easy-to-predict branches, the authors adopted a straightforward approach by examining the bias of various branches. Any branch with a bias surpassing a predefined threshold was chosen for static prediction, with the actual static prediction set to the direction indicated by the bias (taken or not-taken). This strategy proved effective, as highly biased branches are generally easy to predict for any dynamic predictor.

Conversely, determining hard-to-predict branches posed a more intricate challenge. Since the set of hard-to-predict branches varies with the dynamic predictor used, the authors simulated the dynamic predictor in the first phase. They evaluated the prediction accuracy of the simulated dynamic predictor for each branch and selected branches for static prediction based on biases exceeding their prediction accuracies. They used the dominant biases of these branches as static prediction hints. While this meant the performance of the prediction did not suffer, the methodology goes beyond simple profiling and requires extensive simulation of multiple branch outcomes pre-runtime.

The research discovered that predicting some branches statically reduces destructive aliasing in dynamic predictors. It is especially helpful when the dynamic predictors are simple and both prediction schemes focus on different branch behaviours. For example, a static predictor predicts highly taken branches and a dynamic predictor is more suited for branches taken 50% of the time. However, the outcomes of those branches predicted statically are sometimes importantly correlated to branches that the dynamic predictors are predicting, and as such reduce the dynamic performance on those branches by limiting training information. While statically aiding dynamic predictors was shown useful, it is clear future research should aim to avoid removing keep branches that help a dynamic predictor learn.

2.2.4 Using Profiling to Inform the Choice Predictor in a Tournament

In their paper [20] Grunwald, Lindsay, and Zorn propose a novel approach to enhancing branch prediction accuracy through static methods. Hybrid branch prediction, which combines the predictions of multiple single-level or two-level branch predictors, has been widely adopted. However, the prediction-combining hardware, often referred to as the "meta-predictor" or "choice-predictor", can be large, complex, and slow. The authors argue that the combination function can be better performed statically, leveraging prediction hints in the branch instructions.

By incorporating profiling or static analysis, the proposed method sets prediction hints in branch instructions, ensuring that the choice predictor remains static while the actual predictions remain dynamic. This approach mitigates the risk of worst-case performance scenarios. Additionally, the authors highlight that the interference caused by a branch site is limited to a single component predictor, thereby reducing capacity demands.

Empirical evaluations conducted by Grunwald et al. demonstrate the effectiveness of their static hybrid method compared to existing dynamic selection techniques. For instance, results from experiments with realistic benchmarks such as the Instruction Benchmark Suite (IBS) and the SPECint95 suite reveal a lower average miss rate with the static approach. These findings, validated through cross-validation methodologies, underscore the efficacy of the proposed static hybrid predictor.

One key advantage of the static hybrid method is its reduced hardware complexity compared to dynamic-hybrid predictors. Since component selection is performed statically, fewer hardware resources are required. Moreover, as each branch is consistently predicted by the same component, only one component needs to be updated, thereby enhancing the capacity of the hybrid predictor.

However, the static approach presents challenges, including the complexity of the profiling mechanism and the potential for reduced effectiveness due to poor training. Furthermore, static component assignments may not adapt optimally to programs with varying execution phases. Despite some instances of performance degradation, attributed to improper training during profiling, the overall superiority of the static hybrid method is evident. The authors advocate for profile-based component selection to maximize the benefits of their approach.

2.2.5 Clustering Branches by Behaviour

Vandierendonck and Bosschere [21] looked to exploit behavioural properties of branch instructions to increase branch prediction accuracy. They place branches with similar behavioural properties in the same branch cluster. The branch cluster information is fed into the branch predictor to increase its prediction accuracy.

Under Vandierendonck and Bosschere's clustering scheme, branches are dynamically labeled with a cluster identification which serves as an additional source of information. The dynamic branch predictor can then use the cluster identifier used to index branch prediction tables in a similar way as the program counter and the branch history as seen in Figure 2.13.

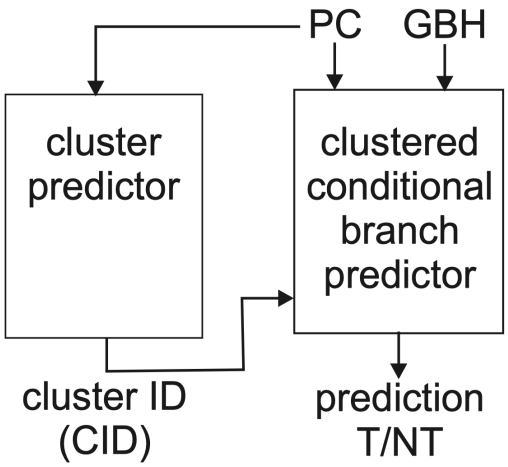


Figure 2.13: Cluster Predictor structure [21]. A dynamic cluster predictor incorporates wider context information about the PC and BTB to assign a CID to the branch. This CID can be used to help inform the prediction of the clustered conditional branch predictor.

The study investigated two clustering approaches based on the taken rate of branches. One used a PC-indexed table to cluster, while the other stored cluster prediction details in the Branch Target Buffer (BTB), a piece of hardware that supplies the target PC value when a program is branching. Vandierendonck and Bosschere found improvements in branch accuracy for a partitioned GShare predictor, with a 3.2% improvement for the PC-indexed scheme and a more substantial 6.3% increase for the BTB scheme.

Notably, the clustering methods used in Vandierendonck and Bosschere's paper are dynamic. This means clusters are constantly changing as the program progresses as they are derived from runtime structures. While this allows the clusters to learn as the program progresses, it limits the information the branch predictor can access. The clusters can only be formed with PC and BTB information, which limits the amount of utility they can provide modern state-of-the-art branch predictors that are far more sophisticated than the GShare predictor augmented by Vandierendonck and Bosschere in 2006.

Ultimately, the authors highlight how clustering branches can be a powerful method to augment current dynamic branch prediction methodologies, and we believe further research into different methods of clustering branches could be a powerful way to provide modern branch predictors with new information.

2.2.6 Manual Branch Hints

The use of manual branch hints by Intel processors [22] provided a means for programmers to influence branch prediction behaviour and optimize code execution. These hints were embedded within conditional branch instructions (Jcc) using specific instruction prefixes, allowing programmers to indicate the most likely code path to be taken at a branch. For instance, the 'Branch Not Taken' hint (2EH prefix) suggested that the branch would likely not be taken, while the 'Branch Taken' hint (3EH prefix) indicated the opposite [23].

By leveraging these hints, programmers could improve the accuracy of branch prediction, thereby enhancing overall program performance. This approach allowed for fine-tuning of branch prediction behaviour based on the expected runtime characteristics of the code. For example, critical loops or conditional statements with known outcomes could be annotated with hints to guide the processor's prediction mechanism.

However, despite their potential benefits, manual branch hints have been phased out in modern

processors. This removal was primarily driven by several factors. Firstly, the effectiveness of manual branch hints was limited by their reliance on human-guided static predictions, which could often be less accurate than dynamic predictions based on runtime behaviour. Additionally, the overhead associated with managing and interpreting these hints outweighed their potential benefits in many cases.

Moreover, the increasing complexity of modern processors, coupled with advancements in dynamic branch prediction algorithms, rendered manual branch hints less relevant. Modern processors now employ sophisticated branch prediction mechanisms, such as branch target buffers (BTBs) and dynamic branch predictors, which can adapt to program behaviour more effectively without relying on static hints.

Furthermore, the use of manual branch hints introduced additional complexity for programmers and compilers. Integrating these hints into code requires careful consideration and could lead to unintended consequences or unpredictable behaviour if not implemented correctly. As a result, the adoption of manual branch hints was relatively low, and their usage declined over time.

In recent processor architectures, manual branch hints have become obsolete, with modern compilers and processors no longer generating or utilizing these hints. Instead, the focus has shifted towards optimizing dynamic branch prediction algorithms and leveraging runtime profiling and feedback mechanisms to improve prediction accuracy.

While manual branch hints once offered a potential means of enhancing branch prediction, their phased-out status reflects the evolution of processor design and the adoption of more sophisticated prediction techniques in modern architectures. These instruction prefixes do not affect modern Intel processors (anything newer than Pentium 4).

2.2.7 Static Transformer-based prediction

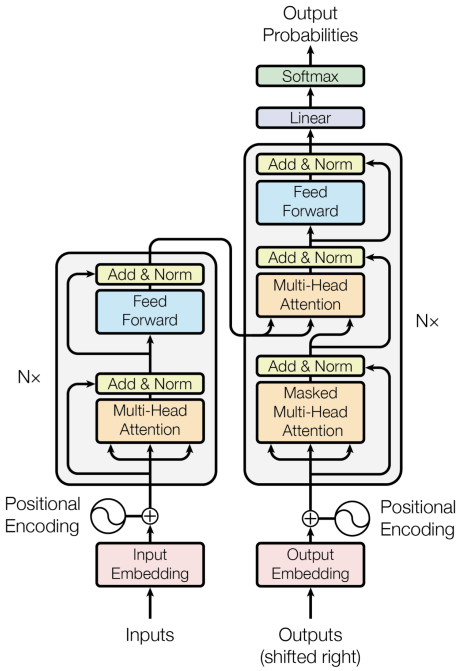


Figure 2.14: Transformer structure [24].

While transformers are Machine Learning prediction methods, Sburlan's 2023 MEng Thesis [3] forms the foundation for the majority of this project. Sburlan experiments with using a transformer network to build a static branch predictor. The project proves a proof of concept that it is possible to get a competitively low miss rate by training a transformer on traces of a program and then using this transformer at runtime to predict the direction of branches. The transformer predictor shows more than 95% accuracy on the training set on some custom synthetic benchmarks. However, the results have some major caveats. The transformer can be trained at compile time or profile-guided optimisation (PGO) time, but the learnt weights must be used to make predictions at runtime.

This means the trained transformer must receive branch outcomes at runtime which is difficult to implement efficiently. Furthermore, modern branch predictors can make multiple predictions on the execution path speculatively and roll back in the event of a mispredict. The rollback mechanism on a transformer is less clear-cut as learnt internal states at runtime would need to be undone.

However, the transformer approach is a very novel way of looking at the branch predictor problem. Transformers are a class of deep learning models that have revolutionized various natural language processing (NLP) and machine learning tasks. Introduced by Vaswani et al. in 2017 [24], transformers have become the backbone of state-of-the-art models like BERT, GPT, and T5.

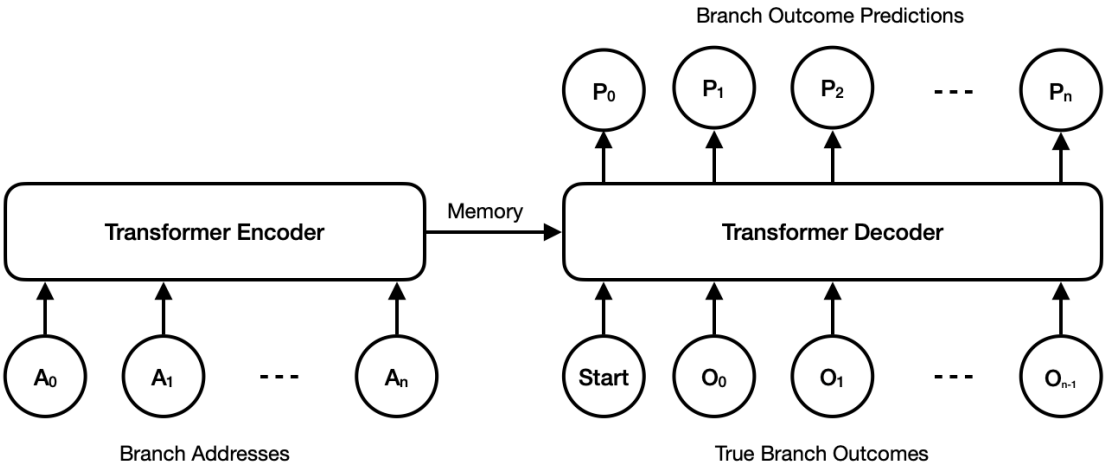


Figure 2.15: Predicting Branch Outcomes with the Transformer Model. The input, A, is the sequence of branch addresses in the path history. The seen branch outcomes, O, is the sequence of known branch outcomes for the input A. P is the sequence of branch predictions. The branch A_n is predicted to have outcome P_n. [3].

Crucially to the work in this thesis, transformers are constructed through a self-attention mechanism that enables them to process input data in parallel, distinguishing them from traditional sequential models. The self-attention mechanism is implemented in normalised Multi-Head Attention units seen in Figure 2.14. These models are designed to handle sequences of tokens, which can represent words, characters, or other units of input data. Self-attention allows each token to consider the importance of every other token in the sequence. Tokens with higher relevance receive more weight during the weighted sum, capturing contextual information and dependencies across the entire sequence and capturing long-range dependencies efficiently.

To utilise transformers, input data is tokenised, breaking it down into units that the model can understand. The tokens are then fed into the transformer, which predicts the next token in the sequence based on its understanding of the relationships and patterns within the input data. This predictive capability is crucial for tasks like language modelling, where the transformer learns to generate coherent and contextually relevant sequences of tokens. The ability to process information in parallel, combined with the predictive nature of token generation, has made transformers the architecture of choice for a wide range of applications in natural language processing and beyond.

Sburlan used the transformer model to build a static branch predictor as seen in Figure 2.15. The transformer's token input is constructed using the lower-order bits of branch addresses. The encoder receives a sequence of recent branch addresses, including the current candidate branch set to be predicted as the final token. Subsequently, the decoder is given a series of recent branch outcomes associated with the historical branch addresses. Predictions are generated by the decoder for each input token, culminating in the last decoder output representing the branch prediction for the candidate branch.

The transformer is trained on only one program, which is the one it is predicting for. This means the transformer can train faster and be smaller than a general-purpose transformer that is trained on a corpus of programs to predict and branch. Such a general-purpose transformer is worth future investigation as it does not require retraining for every new program, but will instead require it to be of a tremendous size to capture the complexities and nuances of all branch behaviour. Sburlan's small transformer model, on the other hand, is trained on traces of the

program it is about to predict. The program is simulated to generate traces of a few branch paths that the transformer learns from to build its understanding of which branches are interdependent. This knowledge allows the transformer to 'overfit' to the program at hand and generate branch predictions for any new unseen trace at runtime.

One issue with this approach is the simulation of the program to generate traces. This simulation must be done at compile time or PGO time and may be prohibitively long to generate multiple traces for large programs. Sburlan uses SimPoints to offer a sophisticated alternative to complete program simulations. SimPoints aim is to identify a compact yet representative collection of samples from the entire program execution, accurately reflecting its characteristics. By simulating these selected samples and appropriately weighting the resulting statistics, we can achieve highly accurate approximations of the full program execution statistics in a significantly reduced time frame.

The project also investigated other ways to build the next-generation branch predictor, including studying the correlation opportunity between the Return Address Stack and branch outcomes, and the correlation opportunity between the Return Address Stack at the time of object allocation and branch outcomes within virtual functions.

2.2.8 Limits of Current Branch Prediction

Exploring the theoretical limits of branch prediction, Chen et al.[1] introduced the Prediction by Partial Matching (PPM) predictor, derived from a compression algorithm that assigns shorter encodings to frequent elements to reduce overall length. The PPM predictor, though resource-intensive, demonstrated optimal performance, the study concluded that most modern predictors are some variation of approximates of the PPM. As such, with sufficient capacity and time, they will approach the theoretical optimum produced by the PPM model. Thus, advancements in branch predictor performance now necessitate the incorporation of more extensive information beyond recent branch history. When new information is introduced, it allows the compression algorithm to adapt and refine its understanding of patterns, optimising the encoding process. Essentially, the algorithm becomes more adept at capturing the nuances and intricacies of the data, resulting in more efficient compression and, by extension, improved branch prediction.

This means the next frontier for branch prediction accuracy is finding the best form of new data to provide the predictor and the optimal way to utilise this new data. To break the PPM limit, we must utilise the information that PPM does not have, highlighting the importance of incorporating new diverse and comprehensive context sources into future branch prediction algorithms.

2.3 Summary

In this chapter, we have set the context for the remainder of this thesis. We have clarified the background ideas driving branch prediction, introduced the relevant architectural units, and explored related work in the field.

Chapter 3

Generating Program Traces

In order to produce colour labels for branches of a program we must have some insight into the branches present in the program from traces. In this chapter, we will explain what our training set of program traces looks like, how it was chosen, and how they were generated. We will also highlight modifications we made to Gem5 to accommodate the use of this training set.

3.1 Sourcing Training Data

For static analysis of a program, we used a training set of traces from the SPEC2006 CPU integer benchmark suite. We chose to analyse the SPEC2006 CPU benchmark suite due to its diverse range of practical programs and being relatively recent. Furthermore, our analysis will focus exclusively on the Integer benchmarks, omitting the Floating Point ones. This decision is based on the observation that Integer benchmarks typically emphasise intricate control flow while Floating Point benchmarks are often more arithmetically intense and are rarely bound by branch predictor accuracy.

The process of obtaining a training set of branches and their takenness for the SPEC2006 integer benchmark suite involved the utilisation of SimPoint [25] and Gem5 [4][5][6]. This process is identical to the one followed by Sburlan [3], and the traces used in this project are identical to those used during their work. SimPoint is a performance evaluation tool used to identify representative portions of a program's execution, known as simulation points, based on statistical analysis of instruction execution patterns. These simulation points serve as reference points for simulation and analysis. While traces can be produced of a program without SimPoint, these take excessively long to produce. Executing benchmarks on a cycle-accurate simulator can lead to significant slowdowns, often increasing the runtime by factors of 1000, resulting in days, weeks, or even longer to complete full benchmarks. However, leveraging SimPoint typically reduces this time by 90-95% [26], while maintaining reasonable accuracy.

The process to generate program traces of the SPEC2006 integer benchmark suite using SimPoint and Gem5:

- **Use BBV to profile programs and generate basic block vectors (BBVs) for them:** A basic block represents a sequential section of code with a defined entry and exit point. Each basic block vector (BBV) logs every basic block entered during program execution, along with the frequency of each block's execution.

- **Pass the BBVs to SimPoint to produce SimPoints:** SimPoint optimises architectural simulations by executing a small segment of a program and extrapolating its overall behaviour. Many programs exhibit phase-based behaviour, where intervals of code execution mimic previous intervals. Detecting and grouping these intervals enables an approximation of the program's overall behaviour by simulating only essential intervals and scaling the results accordingly. This is captured in the BBV offsets and weights produced by Simpoint.

- **Use the SimPoints to generate checkpoints:** Gem5, a cycle-accurate full-system simulator, was utilised to execute the SPEC2006 integer benchmark suite and generate simulation checkpoints for the intervals representative of the program's phases. We use the NonCachingSimpleCPU model as it is the simplest and fastest.

- **Use the checkpoints to produce full traces:** The checkpoints are then resumed using the more detailed out-of-order core, which then generates our custom execution traces. These traces captured the dynamic behaviour of the benchmarks at the instruction level, providing valuable insights into program execution dynamics, memory accesses, and branch behaviour.
- **Extract key information from traces:** The execution traces obtained from Gem5 were then processed to extract information regarding branch instructions and their corresponding outcomes (taken or not taken). This process involved parsing the execution traces, identifying branch instructions based on their opcode and operands, and recording the outcome of each branch (taken or not taken) during simulation. While the work by Sburlan required more detailed trace analytics, our work focuses exclusively on program flow and does not require many features of the more detailed trace.

The resulting dataset comprised a comprehensive collection of branches encountered during the execution of the SPEC2006 integer benchmarks, along with their associated outcomes. Figure 3.1 demonstrates an example trace in the set. The dataset served as the training set for subsequent analysis and evaluation of branch prediction strategies, providing valuable empirical data for assessing prediction accuracy and performance. However, it is important to note that this dataset covers only a subset of all possible branches in the program. The SPEC2006 integer benchmark suite includes a test set that we do not utilise in this study. This test set contains numerous unseen branches and trace paths that remain untaken during the simulation. Thus, while our training set offers insights into the behaviour of branches encountered during simulation, it cannot fully represent the diversity of branch patterns present in the entire program.

However, our goal is to uncover insight from this test set behaviour to help a dynamic branch predictor on all branches of the program.

tick — **u64**	**disassembly** — **str**	**inst_addr** — **u64**	**inst_rel_addr** —	... ...	**call** — **bool**	**taken** — **bool**	**ras_rel** — **list[str]**
10508212000	b <L1>	2688572	handle_compress+2c58	... ...	false	true	["compressStream+2a0]
10508213000	b.cc <L2>	2688620	handle_compress+2c88	... ...	false	true	["compressStream+2a0]
...	...	...	...	...	...	...	...
15813264000	b.hi <L3>	2678792	handle_compress+624	... ...	false	true	["compressStream+2a0]

Figure 3.1: Example Program Branch Trace

3.2 Modifications to Gem5 to use the Traces

The training set was generated on a modified version of Gem5 with custom instructions that were used for statistics logging in the original project. To use both the traces and associated binaries we needed to modify our version of Gem5 to account for this. We added an extra pseudo-instruction that captures the logging instruction during runtime and skips it. Furthermore, the traces were generated using a simulated TAGE branch predictor, but we do not need much of the extra simulated hardware from the traces including the branch predictor results during tracing. Producing a training set akin to the one we use for real-world applications would be much lighter weight and faster than in the work by Sburlan.

3.3 Summary

We demonstrated what our training set of program traces looks like, how they were chosen, how they were generated, and what measures we took to ensure we can use them correctly.

Chapter 4

Augmenting a Dynamic Branch Predictor

We set out to design a novel branch predictor that uses the extra information from colour labels to make better decisions by reducing destructive aliasing. In this chapter, we will explain how our new branch predictor architecture works and how it combines dynamic prediction with provided static instruction colour labels.

4.1 Predictor Design Inspiration and Colour Label Format

We based the design of our custom branch predictor on the state-of-the-art tournament branch predictor [11]. The architecture of this predictor has been discussed prior in the 'Background' section of this thesis. A tournament predictor makes dynamic predictions, however, assuming we have the hardware necessary, we wanted to integrate colour information into the tournament predictor. The colour labels will be used for separating branches that may interfere destructively. They will be generated from a training set of traces of the program we are running. This assignment of colour labels, or clustering, can be done in many ways we discuss in depth later in this thesis. However, to provide some insight, it can be by grouping branches by their behaviour (e.g. takenness), how important they are in informing each other's behaviour (e.g. the attention between them in a transformer), or alternative methods informed by other hardware (such as the Return Address Stack) beyond the scope of this project.

We designed our branch predictor to accept two levels of colour information per branch. This means the predictor is fed one colour for the local level and one colour for the global level. While both these colours could be any number of bits, we use two colours (1 bit) for local and four colours (2 bits) for global. We also noted that not all branches will have colours assigned to them as many will not be in the training set. These unknown branches will be given colour 0 at runtime.

The colour labels are generated by clustering/colouring algorithms at compile time (statically) on the training set of traces. The clustering algorithm can assign the colour 0 to branches if desired, or it can not save the 0 colour only for unseen branches and only assign non-zero colours. Our branch predictor was designed to allow maximum flexibility in optimising the clustering algorithm. We see in Figure 4.1 The clustering algorithms for local and global have assigned colour 0 to branches in the training set (seen at compile time).

However, in some programs colour information may not be very helpful or be detrimental to predictions if clustering is done poorly. This is mainly an issue for global colour labels, as the more simple local predictor gets replaced by the global predictor once it starts under-performing. We ensured colour information could only help the predictor and poor colour information from bad clustering algorithms does not hurt performance much.

When is Branch First Seen?	Program Counter	Disassembly	Local Colour	Global Colour
Runtime	0x4000a1	bne 0x4000b0	0	00
Compile Time	0x4000a5	beq 0x4000b4	1	01
Compile Time	0x4000a9	bgt 0x4000b8	1	00
Runtime	0x4000b1	b 0x4000c0	0	00
Compile Time	0x4000ad	blt 0x4000bc	0	11
Compile Time	0x4000b5	bne 0x4000c4	0	01
Compile Time	0x4000b9	beq 0x4000c8	1	10
Compile Time	0x4000bd	bgt 0x4000cc	0	11
Runtime	0x4000c1	blt 0x4000d0	0	00
Compile Time	0x4000c5	b 0x4000d4	0	01

Figure 4.1: Example Branch Instructions with Local and Global Colouring

4.2 Augmenting the Local Predictor with Colour Information

We augmented the local predictor to maintain its memory footprint as seen in Figure 4.2. Similarly to the normal local predictor, we take the least significant bits of the PC to index the Local History table but we swapped the most significant of those PC bits with the local colour label. This means our modified local predictor has the same size as the original. The 1-bit colour labels effectively partition the local History table into two parts. Branches with colour label 1 can never update an entry in the Local History table starting with a 0 and hence belonging to branches with colour label 0. This prevents any inference between branches of colour 0 and colour 1.

However, we should note each of the two partitions is half the size of the old Local History table. So, if colour labels are assigned poorly and the local colour information is not more useful than the most significant PC bits, the local predictions will get worse as we have fewer bits from the PC being used.

However, we were confident in our ability to generate local colour labels powerful enough to improve performance as the local predictor is simple enough that we can model it statistically. Furthermore, poor local predictor performance means the global predictor takes over from it sooner so we have less to lose in the event of poorly assigned local colour labels.

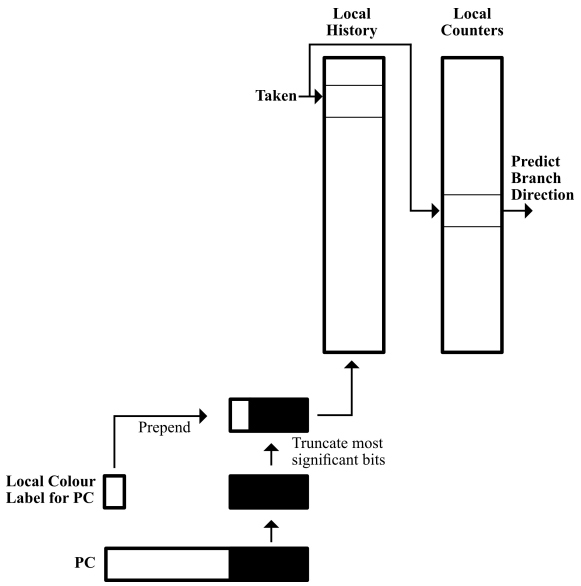


Figure 4.2: Augmented local predictor architecture.

4.3 Augmenting the Global Predictor with Colour Information

We had no guarantees that the clustering algorithm would provide useful global colour labels that may be destructive for predictor performance. The global predictor replaces the local predictor once it warms up and typically runs the remainder of the program. This means it is often responsible for more predictions than the local predictor and has no performant fallback if it starts predicting poorly. We augmented the global predictor to protect against this worst-case using a deeper tournament structure.

The augmented global predictor is fed colour labels for the branch. Uncoloured branches are automatically marked as colour 0. Branches with label 0 are treated differently from other branches in our novel branch predictor.

Every colour label has its own Global History that tracks the taken history for branches marked with that specific colour. When the global predictor encounters a branch with colour C, it looks up the Global History Reg for colour C and colour 0. In the case that C=0, seen in Figure 4.3, then both these Global Histories are identical and only Global History for colour 0 is needed.

We then prepend the colour labels C and 0 to the corresponding Global History Reg, as seen in Figure 4.4, which functions as a selector mechanism. This is then used to index into the Global Counters table and produce two different predictions. One prediction assumes the branch had colour 0 and the other assumes the branch had the colour assigned to it, C. Unseen branches and branches marked with colour 0 produce the same prediction for both as C=0.

Our design of the global branch predictor means that unseen branches and branches with colour 0 use the same global predictor mechanism as proposed by McFarling [11]. The Global History for C=0 is updated by all branches like the original global predictor. Likewise, all the Global Counters that start with the 0 colour label are updated by all branches. Unseen branches can use this to produce a global prediction that is based on all branch history and clustering algorithms can use the colour label 0 to mark branches that may benefit from not having alternate colours even if they have been seen in the training traces.

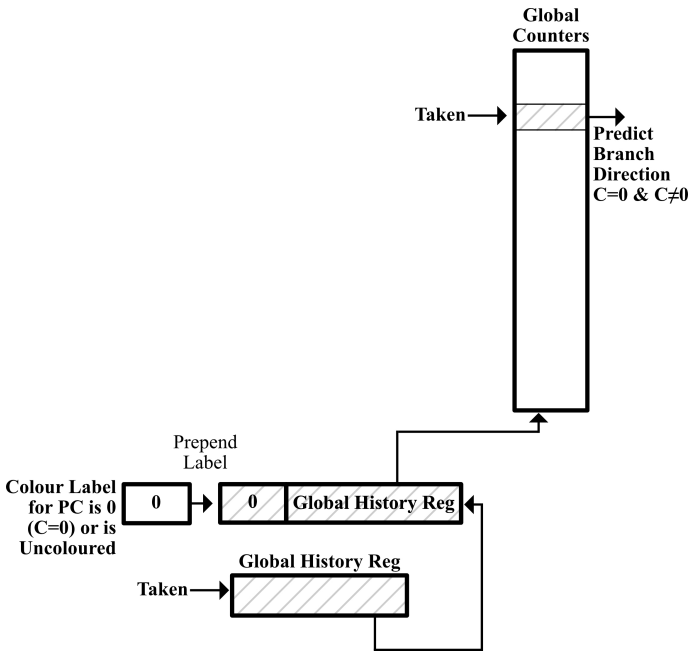


Figure 4.3: Augmented global predictor architecture when faced with a branch that has colour 0 or does not have a colour assigned to it.

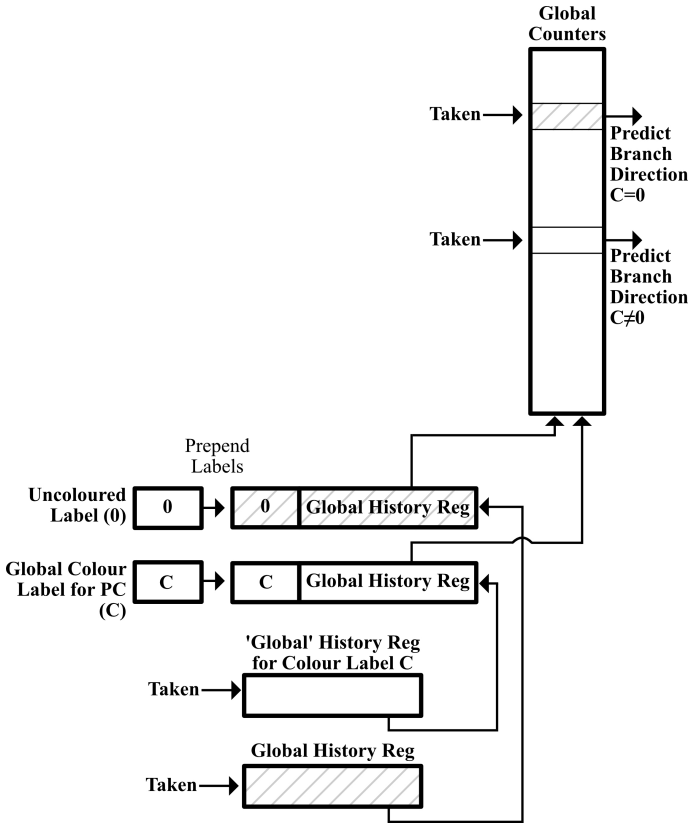


Figure 4.4: Augmented global predictor architecture when faced with a branch that has a non-zero colour assigned to it.

We architected our modifications based on ideas seen in previous research by Patil and Emer [19] that try a significantly more basic strategy to combine static and dynamic predictors. Their shortcomings suggest we should aim to avoid removing branches from a dynamic branch predictor learning set unless we are certain it is not useful. By including the 0 path that predicts ignoring colour information, we ensure we always have a fallback that learns from all branches like normal. However, in the case where the colour labels are useful, we also produce predictions using them and learning from a subset of branches.

4.3.1 Adapting the Choice Predictor for a Deeper Tournament

Our modification of the global predictor means that it produces two predictions. One prediction uses the colour label assigned to it, C≠0, and the other ignores the colour label assigned to it and instead uses C=0. We decide at runtime whether the predictions will be better with or without using the assigned colour labels.

The original tournament predictor uses a secondary internal predictor to determine when to use the local predictor and when to use the global predictor. We extended the functionality of this choice predictor to choose between using the C=0 or C≠0 global prediction. Instead of indexing the choice predictor using the PC as proposed by McFarling, we index using the Global Histories and the colour labels as selectors. When the true outcome for a predicted branch is revealed, the choice predictor compares it to what the augmented local predictor predicted and what the uncoloured C=0 global predictor predicted and modifies accordingly the counter indexed by the C=0 Global History Reg and C=0 label. This choice update is analogous to the original tournament choice prediction mechanism. The choice predictor then compares the true outcome of the branch to what the uncoloured C=0 global predictor predicted and what the coloured C≠0 global predictor predicted and modifies accordingly the counter indexed by the C≠0 Global History Reg and C≠0 label. If the branch was uncoloured or C=0, then this second comparison never occurs.

The augmented choice predictor runs inside our augmented tournament predictor to choose between using local and global predictors based on their historical performance. If it chooses the global predictor, it then compares the historical performance of the global predictor when it uses

the colour labels and when it ignores them, to decide if the colour label assigned to this branch is useful. Furthermore, we highlight that this mechanism can differentiate between colours, so our choice predictor can identify cases where branches marked C=1 benefit by using their label, but branches marked C=2 are better off being predicted with the original C=0 global predictor and ignoring the label.

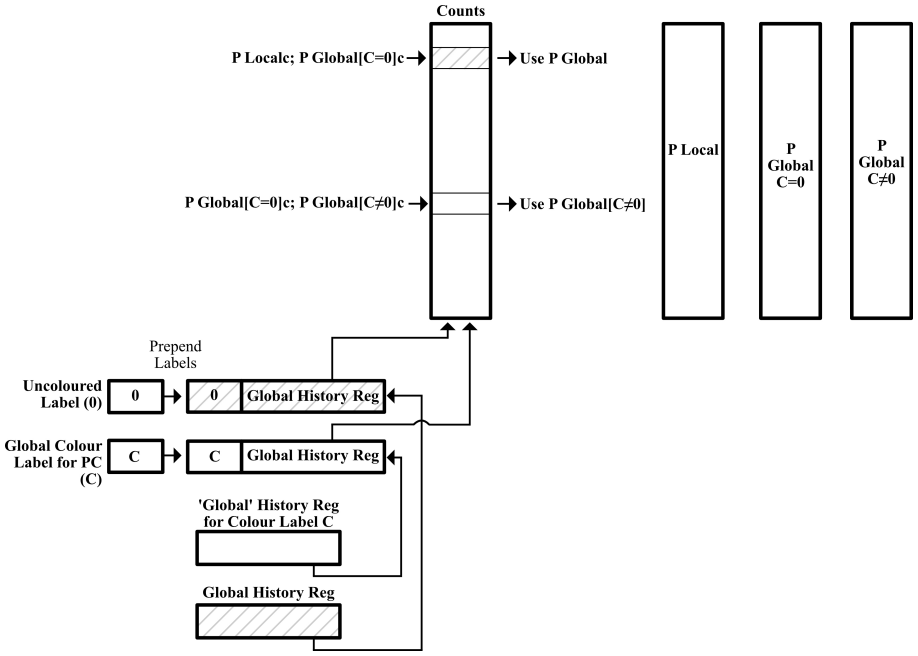


Figure 4.5: Augmented choice predictor architecture.

It should be noted that the memory requirements of our augmented global predictor are increased to account for the colours. However, if we are confident that our clustering algorithm produces useful colours then the deeper tournament structure can be removed and always return the coloured prediction. This would make the space requirements of the augmented predictor identical to the McFarling predictor [11]. We keep the deeper tournament to better assess the effectiveness of different clustering algorithms later in this thesis.

4.4 Implementing the Augmented Branch Predictor in the Gem5 Simulator

We then implemented our custom branch predictor architecture in Gem5 and integrated it with the rest of the microarchitecture simulation. This included the full physical structure of the predictor, speculatively updated histories, and comprehensive squash and rollback mechanisms. Additionally, we developed custom performance statistics logging to thoroughly analyse the behaviour and efficacy of our predictor design under various workloads and conditions.

4.5 Summary

We have designed a custom branch predictor that can make dynamic decisions while simultaneously being informed by colour labels.

Chapter 5

Assigning Colours using Branch Takenness

In this chapter, we will highlight some issues faced by the current tournament branch predictors and how we can generate local colour labels using the takenness of branches to alleviate these problems in our novel branch predictor. We will also explore the mathematical underpinnings behind why and when the clusters improve prediction accuracy. Finally, we will describe the specifics of the clustering algorithm we use to produce the colour labels.

5.1 Rationale

One issue with the modern unaugmented tournament predictor is the aliasing of multiple branches to the same local history register in the local predictor. This aliasing is because we only use the least significant bits from the PC to index into the history table, so every branch in the program does not get its local history perfectly tracked. Aliasing can manifest as a problem in two cases we study further:

1. Two branches that alias to the same history register are being seen by the program at the same time. Therefore, the clash forces history to be shared and corrupted, which makes the predictions for both branches worse. In the example 5.1, we see how if we use only the 2 least significant bits from the PC to index the history table, the branch `if (x % 3 == 0)` and the branch `if (i % 2 == 0)` both alias to the entry in the history table (00). This PC in the examples are strictly demonstrative and will not be accurate when translated to assembly.

```
1   void foo(int x) {
2       if (x % 3 == 0) {                                   // PC: 0000
3           printf("x is divisible by 3\n");                // PC: 0001
4       } else {
5           printf("x is not divisible by 3\n");            // PC: 0010
6       }
7   }
8
9   int main() {
10      int i;
11
12      for (i = 0; i < 100; i++) {                         // PC: 0011
13          if (i % 2 == 0) {                               // PC: 0100
14              foo(i);                                     // PC: 0101
15          } else {
16              printf("i is not divisible by 2\n");        // PC: 0110
17          }
18      }
19
20      return 0;
21  }
22
```

Listing 5.1: Local Branches Aliasing Simultaneously using the 2 Least Significant PC Bits

Without aliasing, the historical takenness of `if (i % 2 == 0)` would be stored in its own history entry. Histories that have the least significant bit 0 would point to a counter table entry predicting the next branch is taken. Conversely, histories that have the least significant bit 1 would point to a counter table entry predicting the next branch is not taken. This requires just 2 bits of history to encode the pattern. Likewise, with `if (i % 3 == 0)`, we need only 3 bits of history to encode the branch pattern. However, with aliasing we require the 6 bits, to capture the pattern. With more complicated branches that are more independent of each other, this extra cost of aliasing increases further. The more bits of history we need to capture a pattern, the longer it takes for the history to 'fill up' and for the relevant counters to be saturated in the correct direction. Furthermore, if the aliased branch pattern is too complex to fit in the history bits used in the local predictor, as is often the case, then the local predictor cannot make correct predictions.

Gem5's default implementation of the local predictor uses 11 bits of the PC to index the local history. So branches with the same bottom 11 bits will alias to the same history.

2. Two branches that alias to the same history register are being seen by the program at different times. By the pigeonhole principle [27], this is a very common and necessary alias when the local history table is smaller than the number of unique branch instructions.

```
1
2   int main() {
3       int i;
4
5       for (i = 0; i < 100; i++) {                          // PC: 0000
6           for (x = 0; x < 100; x++) {                      // PC: 0001
7               if (x % 2 == 0) {                            // PC: 0010
8                   printf("x is divisible by 2\n");         // PC: 0011
9               }
10              ...
11          }
12          ...
13          for (y = 0; y < 100; y++) {                      // PC: 1101
14              if (y % 3 == 0) {                            // PC: 1110
15                  printf("y is divisible by 3\n");         // PC: 1111
16              }
17              ...
18          }
19          ...
20      }
21
22      return 0;
23  }
24
```

Listing 5.2: Local Branches Aliasing At Different Times using the 2 Least Significant PC Bits

In this situation 5.2, the local history register for `if (x % 2 == 0)` is repeatedly inherited by `if (y % 3 == 0)` and then handed back. The local branch predictor keeps building on the old history with taken patterns of the new branch until the history is saturated and fully aligns with the new branch. During this warm-up phase, the old history impacts the counter registers that are updated and slows down the saturation of the correct counters for the new branch's history patterns. This warm-up time is quite short for local predictors and much larger for global predictors, but reducing it in any capacity would push us closer to a hypothetical perfect predictor.

One way we believe we can alleviate some of the cost of aliasing is by clustering by takenness. We describe takenness to be whether a branch is more frequently taken or not-taken. This means we use one fewer bit from the PC when indexing the local history table and instead use one bit to supply information on the branches takenness in the training set.

The colour labels we generate by clustering branches by takenness are used by the local predictor in our augmented tournament predictor. When a branch is marked as more likely to be taken by its colour label it can only alias in local history with other branches marked in the same colour. Aliasing with branches that are more likely to go the same way reduces the destructive interference.

Furthermore, the partition of the local history table that tracks these more often taken branches will have more 1's than 0's, and the partition of the local history table that tracks these more often not-taken branches will have more 0's than 1's. This means the local counters table has a soft statistical partition where branches marked in the same colour are more likely to produce predictions from the same subset of local counters as other branches with the same colour.

However, this is all assuming we can assign colour labels to confidently say a branch is more often taken or more often not-taken. In practice, this is not always possible because of variations in branch behaviour in the limited training set and the presence of many unseen branches at runtime. To study the effectiveness of clustering branches by takenness and the extent to which we can be confident in the takenness assignment of a branch, we do a statistical analysis.

5.1.1 Analysis of Local Takenness-Based Colour Utility

We studied the statistical benefit of colour labels in depth. Our detailed analysis and mathematical proof on the bounds of utility for our local colouring scheme can be found in Appendix A to not distract from the experimental nature of the remainder of this thesis. Our key findings are:

- Proof that takenness colour labels can provide a statistical advantage for improving the local predictor accuracy in both situations discussed earlier in this chapter.
- Since the local predictor in our augmented tournament maintains the same memory footprint, this can cause extra cost in more aliases as we are using fewer PC bits.
- We show this extra alias cost of our colour labels maintaining the same memory footprint is a function of how much of the program behaviour the training set traces cover.
- We suffer no expected extra alias cost, however, if our colour labels at run time, including unseen branches automatically assigned colour 0 as specified by the augmented tournament architecture, are split 50/50.
- We cannot confidently say branches with the more often taken colour are taken more than 50% of the time and branches with the more often not-taken colour are taken less than 50% of the time. While these statements are true when we analyse the training set, during runtime unseen branches are assigned colour 0. This shifts the bounds of the branches labelled colour 0. This presence of unseen branches means we can only assume branches with colour label 1 are bounded by the 50% mark. However, we show that this is enough to improve prediction accuracy in great detail in Appendix A.
- Overall, if we achieve close enough to the 50/50 colour assignment split at run time by having a representative and large training set of traces, then a predictor using colour labels will always produce better predictions in the same memory footprint than a local predictor not using them.

Appendix A provides a more rigorous explanation of why colouring helps with aliasing. This analysis informed our implementation of a clustering algorithm to assign colours by takenness.

5.2 Implementation

In the process of clustering branches based on their takenness, we aim to assign each branch in the trace to one of two clusters: 0 or 1. The underlying principle revolves around leveraging the trace behaviour of branches to categorise them into clusters that signify their likelihood of being taken or not taken.

The clustering process starts by collecting data from the trace, including the branch addresses and their corresponding outcomes (taken or not taken). Additionally, the weights associated with each simulation point are considered, reflecting the relative frequency of execution for different parts of the benchmark program. We can then categorise branches based on whether they are more frequently taken or not taken.

The crucial decision-making step in the clustering process comes when determining whether a branch more often taken should be assigned to cluster 0 or cluster 1. To ensure a balanced distribution of branches across the clusters, we strategically assign clusters such that the majority

of branches are marked as cluster 1. By doing so, we aim to maximise the statistical 'room' for unseen branches to be marked as cluster 0 without causing clashes with the observed branches. This ensures we get as close to the target 50/50 split we proved we need to aim for in the previous section.

5.3 Results

We generated local colour labels for the SPEC2006 CPU benchmarks using the training set traces and fed these to the Gem5 simulator with our augmented tournament branch predictor implementation. The changes in prediction performance, measured in Figure 5.1 as 'Misses-per-Kilo-Instruction', represent how many instructions our simulated microarchitecture miss-predicts per thousand it commits.

Benchmark	Default Tournament Misses-per-Kilo-Instruction	Augmented Tournament Misses-per-Kilo-Instruction *Local Takenness-based Labels*	Percentage Improvement
sjeng	14.09	13.69	2.85
mcf	20.63	20.62	0.01
hmmer	1.71	1.71	0.00
h264ref	1.57	1.55	0.94
gobmk	19.68	19.15	2.70
bzip2	6.59	6.59	0.01
perlbench	6.04	5.71	5.50
xalancbmk	4.22	4.23	-0.25
astar	37.25	37.25	0.00
gcc	7.49	7.32	2.22
omnetpp	5.81	5.66	2.59
Average	11.37	11.22	1.27

Figure 5.1: Performance difference between the original tournament branch predictor and a colour augmented tournament branch predictor fed with takenness local colour labels. Lower Misses-per-Kilo-Instruction is better.

The local takenness-based colour labels improve the average performance across SPEC2006 versus the state-of-the-art tournament despite maintaining the same memory footprint. The labels never hurt performance in any individual benchmark except for xalancbmk, where we believe the training set traces were not representative enough to get the 50/50 split we require to minimise the extra alias cost of keeping the memory footprint the same. The global predictor functioned the same as the default tournament predictor.

5.4 Summary

In this chapter, we addressed several issues encountered by current state-of-the-art branch predictors and examined how generating colour labels could mitigate these challenges. We explored the mathematical foundations of why and when the takenness-based clusters improve prediction accuracy. Finally, we described the specifics of the clustering algorithm we used to produce the colour labels.

Chapter 6

Assigning Colours using a Transformer

In this chapter, we will highlight some issues faced by current state-of-the-art branch predictors and how we can generate global colour labels using the learnt attention of a transformer to combat these in our novel predictor. We will describe the specifics of the clustering algorithm we use to produce the colour labels.

6.1 Rationale

In research by Sburlan [3], they propose a static transformer branch predictor architecture. This transformer was trained on traces of the program at compile time and at runtime it would hypothetically be fed with the outcomes of branch patterns.

While the transformer produced comparable accuracy results to TAGE-SC-L 64KB, it is unrealistic to implement in practice. The biggest issue is that the trained transformer weights and architecture would need to be implemented in hardware for it to produce predictions at runtime fast enough. This is a huge power and silicon cost that would need to somehow change configurations between program runs to reflect the learnt transformer weights of that particular program.

However, we saw merit in the natural language processing methodical and designed a clustering algorithm to condense the learnt information of a transformer into colour labels.

A code pattern that modern branch predictors struggle to deal with is what we describe as 'phased' code. The state-of-the-art dynamic branch predictors have a limited memory about the history of a program so when they encounter new branches they must forget older information to learn the latest patterns.

Example 6.1, demonstrates an example of 'phased' code that could pose a problem to modern branch predictors. During Phase 1, the branch predictor encounters highly correlated branches, which it can efficiently predict based on its learned patterns. As the program progresses through Phase 1, the branch predictor builds a strong understanding of the branch patterns associated with this phase, optimising its predictions for future iterations of similar branches.

However, Phase 2 presents a significant challenge for the branch predictor. The introduction of random branching based on memory values disrupts the predictor's learned patterns. With each new, uncorrelated branch encountered in Phase 2, the branch predictor must overwrite previously learned patterns to accommodate the new information. This process results in the corruption of the predictor's memory regarding Phase 1 branches. When Phase 3 begins, the branch predictor is forced to relearn the patterns it previously encountered in Phase 1. However, due to the corruption of its memory during Phase 2, the predictor makes numerous mispredictions as it attempts to reestablish its understanding of the Phase 1 branch patterns. While Phase 2 in the example consists of completely random branches, it could also include branches uncorrelated with Phases 1 and 3. The key challenge lies in ensuring that the branch predictor retains essential information from Phase 1 to improve predictions during Phase 3.

To address this challenge, we propose leveraging a static natural language processing approach to identify and understand the occurrence of these distinct phases within the program. By identifying the boundaries between phases and recognising the characteristics of each phase, we can

```
        // Phase 1: Highly correlated branches
        for (int i = 0; i < 1000; i++) {
            if (i % 2 == 0) {
                ...
            } else {
                ...
            }
        }

        // Phase 2: Random branching based on memory values
        int *arr = (int *)malloc(1000 * sizeof(int));
        for (int i = 0; i < 1000; i++) {
            arr[i] = rand();
        }
        for (int i = 0; i < 1000; i++) {
            if (arr[i] % 2 == 0) {
                ...
            } else {
                ...
            }
        }

        // Phase 3: Highly correlated branches again
        for (int i = 0; i < 1000; i++) {
            if (i % 2 == 0) {
                ...
            } else {
                ...
            }
        }
```

Listing 6.1: An example program with 3 distinct phases

provide this information to the dynamic branch predictor. Armed with this contextual understanding, the predictor can prioritise retaining crucial information from Phase 1, even when faced with disruptive phases like Phase 2, thereby enhancing overall prediction accuracy and efficiency.

6.2 Implementation

Unlike traditional branch predictors, transformers utilise a fundamentally different approach to prediction, drawing inspiration from the field of natural language processing. Transformers excel at capturing long-range dependencies and making nuanced contextual decisions, enabling them to overcome the challenges posed by phased code. One key advantage of transformers lies in their ability to analyse the program's execution history comprehensively. By considering a broader context and capturing intricate dependencies between program instructions, we believe transformers can make more informed predictions, even in the presence of disruptive phases.

Attention mechanisms in transformers facilitate the model's ability to focus on different parts of the input sequence when generating an output. For every token, or in our case branch, in the sequence, three distinct vectors are derived using learned weight matrices: the query (Q), key (K), and value (V) vectors. The query vector represents the token's request for information from other words, the key vector serves as an identifier to determine the relevance of other words in response to the query, and the value vector contains the actual information pertaining to the word.

The attention score, which determines the focus a token should give to another, is computed by taking the dot product of their query and key vectors, followed by scaling with a factor dependent on the dimension of the key vectors:

$$\text{score}(A, B) = \frac{\text{Q}_A \cdot \text{K}_B}{\sqrt{d_k}}$$

where d_k is the dimension of the key vectors. These scores are then normalised using a softmax function to produce a probability distribution, ensuring that the scores sum to one:

$$\text{attention weight}(A, B) = \text{softmax}(\text{score}(A, B))$$

This attention weight is used, along with the value vectors of all tokens in the sequence, to produce the prediction for the next token. However, the attention weights alone can provide insights into what the transformer has learnt. It tells us what branches the transformer used to make its prediction of the current branch, which can help us understand which branches influence each other and which do not. This, ultimately, is a gateway to uncover insight about the phases of a program we are looking for.

To harness the power of transformers for branch prediction, we developed a clustering algorithm that utilised the attention mechanism inherent in transformer architectures. The algorithm first trained a transformer on training set traces of a program, capturing intricate dependencies between program instructions. This transformer was largely identical to the architecture proposed by Sburlan, with some key modifications to allow better attention extraction. Namely, we remove the masking operation to allow a one-to-one mapping out of the encoded state back to branches. Specifically, we use:

- sequence length of 512 tokens
- 8 encoder layers and 8 decoder layers
- 8 head Multi-head Attention Modules
- 5 Epochs

Central to our approach was the extraction of the first layer of attention for every input. Traditionally, attention is applied sequentially in multiple layers, however, we focused solely on the first layer (Layer 0) attention to have the least abstracted view of learnt patterns. By merging these first layer attentions into one global attention map and by removing the masking operation, we ensured that branches encountered multiple times were accurately represented. We also ensure the global attention map is symmetric, such that a branch A being attended to by another branch B, contributes to A's own attention of the other branch B as the transformer has learnt they are connected. The attention matrix generated by Sburlan in Figure 6.1 shows the attention between tokens in a trace of the gobmk benchmark. We noted they find attention between tokens, which do not correspond to branch addresses directly due to the masking operation. Furthermore, this attention matrix is of one particular trace, so branches will be present multiple times in the matrix and not all traces are accounted for. Our global attention map for gobmk, seen in Figure 6.2, is indexed on branch addresses and multiple attention matrices from all traces are combined to form a global attention map.

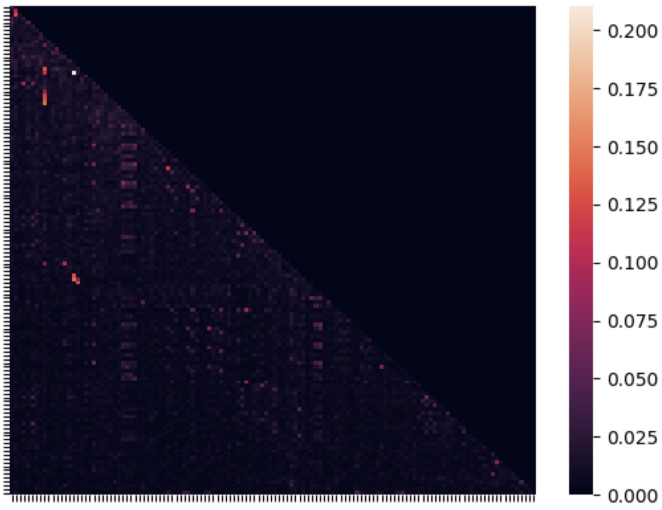


Figure 6.1: Cropped attention matrix in the SPEC2006 gobmk benchmark using the original architecture from Sburlan [3].

The resulting attention matrix formed the basis of our graph representation, where attention weights served as edges between nodes representing program branches. We utilised a community detection algorithm, specifically the fast greedy algorithm, to identify clusters within the graph.

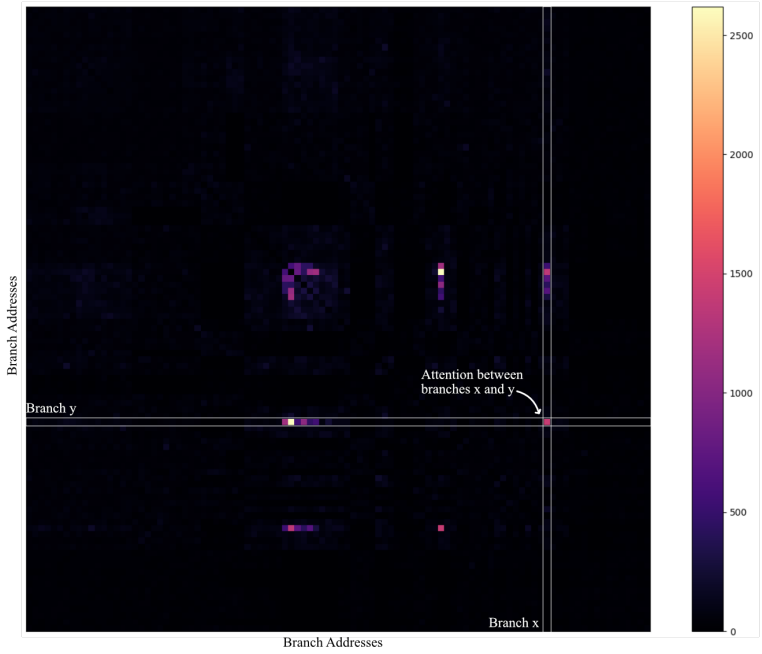


Figure 6.2: Cropped global attention map in the SPEC2006 gobmk.

The process began by converting the attention matrix to a weighted graph using the attention weights as edge weights, as exemplified in Figure 6.3. We then applied the fast greedy algorithm [28] to detect communities within the graph. The fast greedy algorithm starts with only links between highly connected nodes. Then, the algorithm iteratively samples random links that improve the modularity of the subnetwork and adds them. This iterative process is repeated as long as the modularity keeps improving. The process can be seen run on the example in Figure 6.4. One notable aspect of our approach was the ability to define the desired number of clusters, allowing us to tailor the clustering process to the specific requirements of the program under analysis. The fast greedy algorithm is a hierarchical agglomeration algorithm making it faster than many competing algorithms with the key advantage of being able to segment large graphs into any number of communities due to its iterative nature. The algorithm's greedy nature allows it to detect communities on much larger graphs in almost linear time on sparse graphs like the ones we produced with our global attention maps. However, we must also consider the greedy nature also means it is not guaranteed to produce the optimal clusters. We believe slightly sub-optimal communities were acceptable to facilitate reasonable compute times on the large graphs we produce. In our case, we aimed to identify three distinct clusters corresponding to the 2 bits of colour we aim to generate.

6.3 Results

We generated global colour labels for the SPEC2006 CPU benchmarks using the clustering algorithm described. The changes in prediction performance are shown in Figure 6.5.

We see that on average there is a decrease in predictor performance. The majority of this deficit is from the astar benchmark. However, we do see several benchmarks with tangible improvements, suggesting the approach holds merit.

While we designed the augmented tournament global predictor architecture to be resilient to bad colour labels, we see that competition between the coloured and uncoloured global predictor has led to the coloured predictor being chosen when it should not have been. We hypothesise this is because the colour was useful in one stage of the program but not for the next stage. When it reaches the second stage the choice predictor must flip back, this delay in choosing the predictor causes a constant contention between using the colour labels and ignoring them, and so leads to excess misspredicts. This choice predictor delay is not an issue in the local-global tournament of the default tournament predictor as the flip happens very few times and the local predictor is

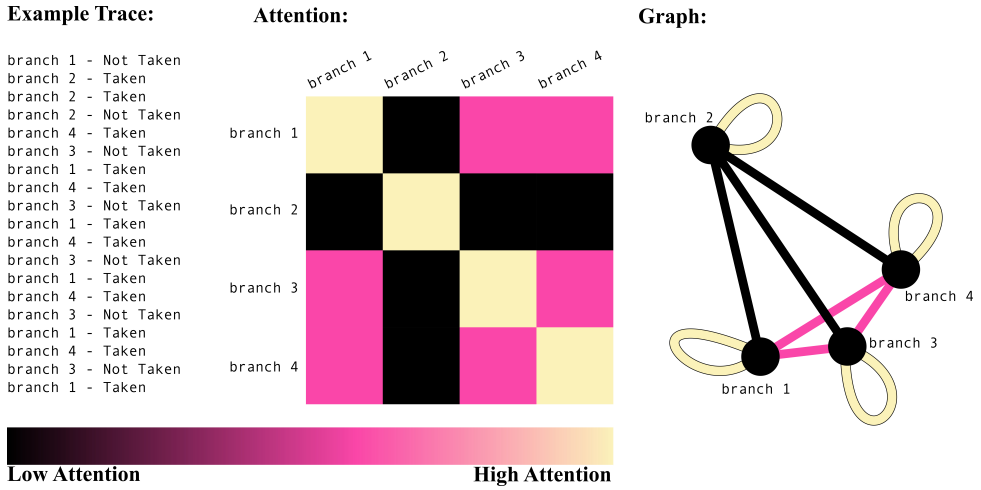


Figure 6.3: Example trace being converted to global attention map and then subsequently into a graph.

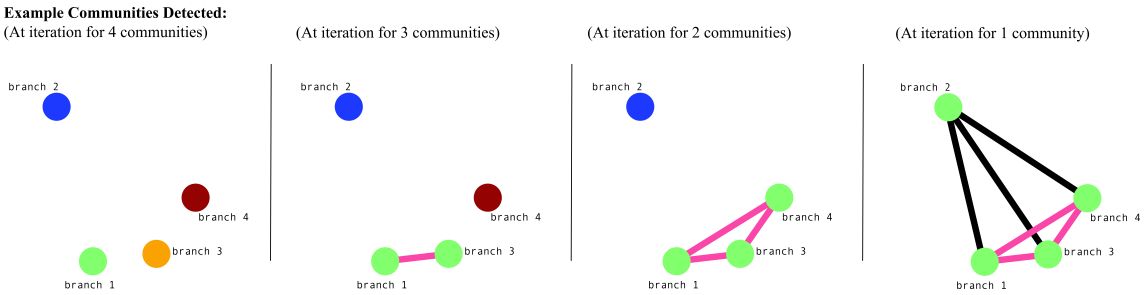


Figure 6.4: Fast greedy community detection on an example attention matrix. Distinct node colours represent unique communities.

only useful while the global warms up. However, if the colour labels can consistently and reliably provide useful information this becomes less of an issue, as the flip happens fewer times.

Another reason we believe some benchmarks suffered a performance loss is because the attention produced by the transformer is too smooth. The transformer has a long 512 sequence length and like most traditional transformers each token attends to other tokens in the sequence it can see. Firstly, a token may have to attend to 511 other tokens, which results in a smoother attention matrix which is more sensitive to training noise. We see this in Figure 6.2, where attention across the entire map is low and not very expressive. While we do see a few fireflies, they do not provide the community detection algorithm with enough information to make informed decisions. Instead, the community detection algorithm tries to infer patterns in the noisy low-attention spaces. This can lead to noise overpowering the community detection stage of the clustering process. Also, this smoothness can be a result of the depth and complexity of the transformer. The first layer becomes less expressive as the complexity of the model increases as the transformer can learn nuanced patterns combining many branches and the more complex these patterns, the more obfuscated they are in Layer 0. The more complex the patterns the less useful they are to the significantly more simple runtime predictors. Secondly, the large sequence length, while necessary for program phases to be uncovered can lead to the transformer attending to patterns that are not translatable to a global predictor. The transformer can see the branch history of the particular branch it is trying to predict and will attend to itself rather than the surrounding correlated branches we need it to attend to identify phases in a program.

6.3.1 Dual Colouring Scheme

Our augmented tournament architecture allows us to feed both transformer-based and takenness-based labels simultaneously. The global predictor uses the former and the local predictor uses the latter. We see the performance benefits of the two colouring schemes complement each other.

Benchmark	Default Tournament Misses-per-Kilo-Instruction	Augmented Tournament Misses-per-Kilo-Instruction *Global Transformer-based Labels*	Percentage Improvement
sjeng	14.09	13.87	1.52
mcf	20.63	20.47	0.74
hmmer	1.71	1.71	0.16
h264ref	1.57	1.56	0.34
gobmk	19.68	19.32	1.85
bzip2	6.59	6.58	0.05
perlbench	6.04	6.18	-2.39
xalancbmk	4.22	4.26	-1.14
astar	37.25	39.62	-6.37
gcc	7.49	7.47	0.14
omnetpp	5.81	5.90	-1.57
Average	11.37	11.54	-1.52

Figure 6.5: Performance difference between the original tournament branch predictor and a colour augmented tournament branch predictor fed with global colour labels derived from the learnt attention of a transformer. Lower Misses-per-Kilo-Instruction is better.

Benchmark	Default Tournament Misses-per-Kilo-Instruction	Augmented Tournament Misses-per-Kilo-Instruction *Global Transformer-based and Local Takenness-based Labels*	Percentage Improvement
sjeng	14.09	13.51	4.11
mcf	20.63	20.47	0.75
hmmer	1.71	1.71	0.17
h264ref	1.57	1.55	1.23
gobmk	19.68	18.83	4.31
bzip2	6.59	6.58	0.06
perlbench	6.04	5.83	3.50
xalancbmk	4.22	4.27	-1.34
astar	37.25	39.62	-6.38
gcc	7.49	7.45	0.41
omnetpp	5.81	5.71	1.77
Average	11.37	11.41	-0.39

Figure 6.6: Performance difference between the original tournament branch predictor and a colour augmented tournament branch predictor fed with takenness local colour labels and transformer derived global colour labels simultaneously. Lower Misses-per-Kilo-Instruction is better.

6.4 Summary

In this chapter, we demonstrated the code phase issues faced by current state-of-the-art branch predictors and showed our method for generating global colour labels using the learnt attention of a transformer.

Chapter 7

Assigning Colours using a Rolling Window Transformer

In this chapter, we will go over a method for improving the transformer-based labels using a rolling window approach.

7.1 Rationale

We sought to improve on the shortcomings of the previous transformer-based clustering algorithm by modifying the traditional approach of processing sequences in transformer-based models by implementing a rolling window mechanism. This decision was motivated by various factors, including the observed success of rolling window techniques in computer vision applications [29, 30], as well as their effectiveness in predicting transient systems, such as the productivity of oil wells [31].

Program branch traces, similarly to transient systems, exhibit a dynamic nature where the branch predictor's memory of past branches is limited. Therefore, we believe our approach to colouring should prioritise clustering optimisation for branches within recent memory, enabling the predictor to make more accurate predictions based on the most relevant branch patterns.

Additionally, we drew inspiration from the Longformer architecture [32], a state-of-the-art transformer model designed to process long sequences efficiently. One of the main limitations of traditional transformer architectures lies in their inability to efficiently process long sequences due to the quadratic scaling of the self-attention operation with sequence length. The Longformer addresses this limitation by adopting a sliding window attention mechanism, resulting in a linear scaling of computational complexity with sequence length. Similarly, our decision to implement a rolling window mechanism reflects a parallel effort to mitigate computational challenges and improve the model's efficiency when processing long sequences, such as entire program traces.

While the Longformer primarily employs a sliding window attention mechanism to mitigate the computational challenges associated with processing long sequences, our adaptation involved using a rolling window approach to process the input sequence instead.

7.2 Implementation

We no longer split traces into 512 token sequences, and instead use a 52 token window into the trace and progress this window 13 tokens at a time. Our 2 bits of colour information should allow the 13 bits of global history to capture roughly 4 times more information, so our window and progress step size reflect this statistical estimation along with concerns on memory consumption. However, we believe the specifics of the window and progression step can be optimised further. We also simplify the architecture of the transformer, as seen in Figures 7.1 7.2, to make the first attention layer more expressive. The remainder of the transformer is identical to a traditional transformer as described in Attention Is All You Need [24] and re-described in context by Sburlan [3] The new architecture is as follows:

- sequence length of 52 tokens
- 4 encoder layers and 4 decoder layers

- 4 head Multi-head Attention Modules
- 6 Epochs

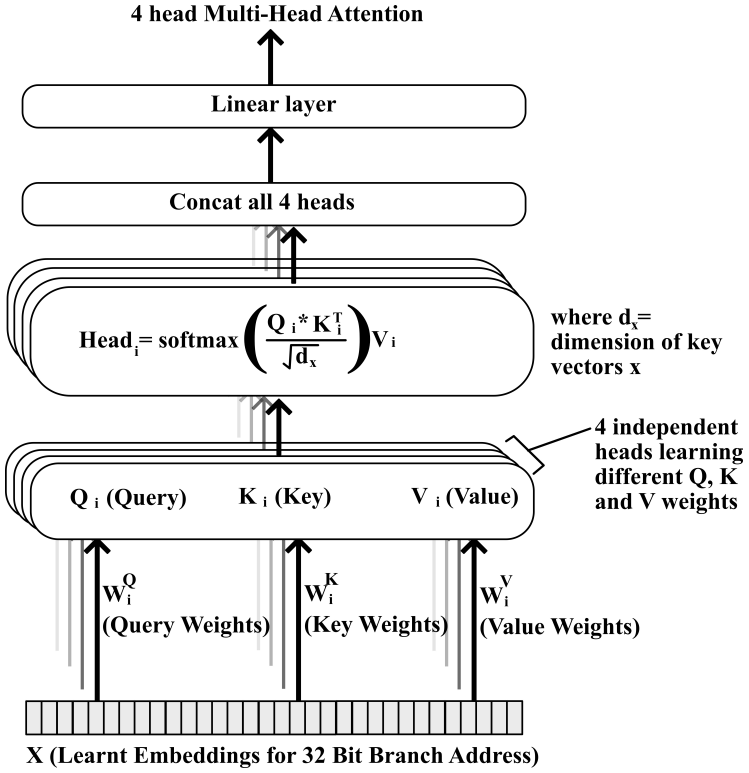


Figure 7.1: A single 4 head multi-attention module used in the rolling window transformer. Architecture based on Attention Is All You Need [24].

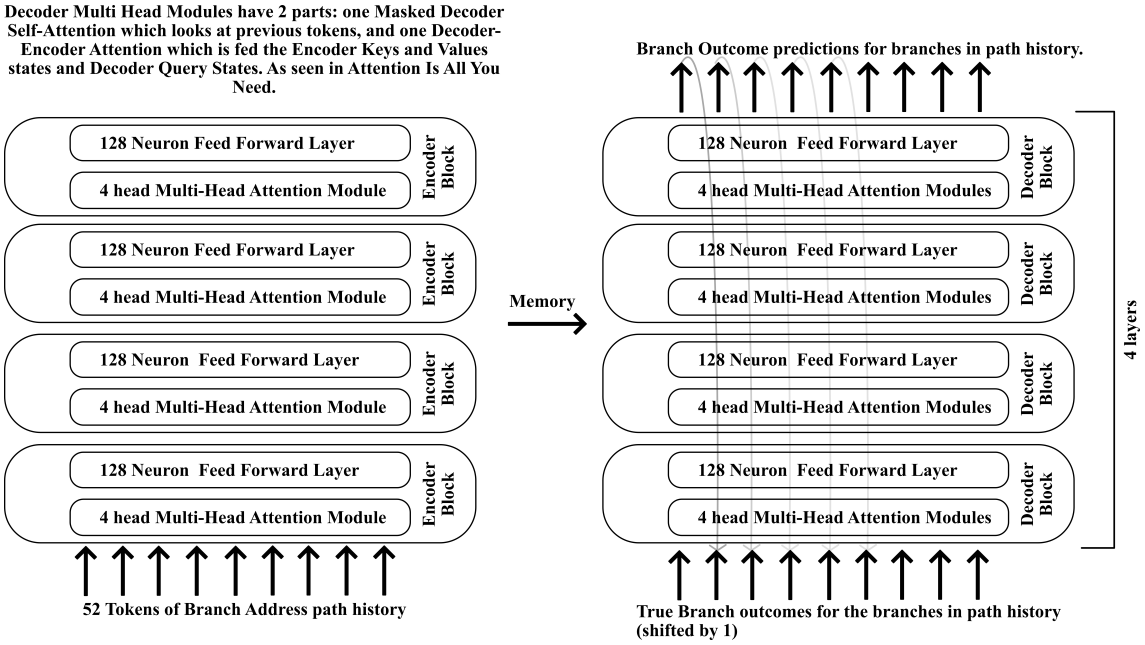
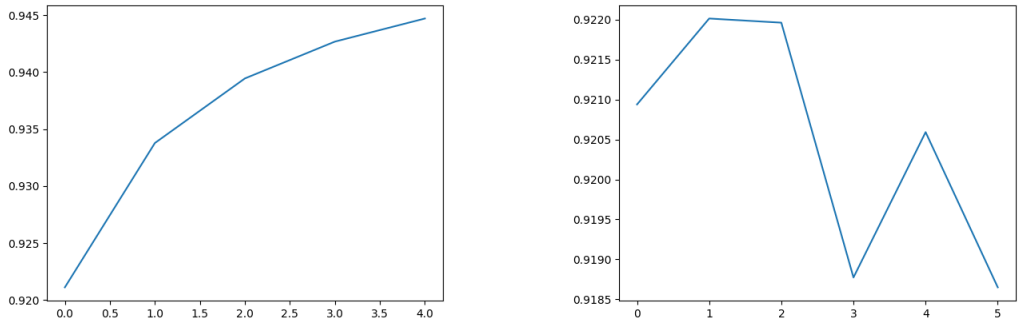


Figure 7.2: The 4 encoder layer and 4 decoder layer architecture of the rolling window transformer. Fewer layers force the first layer Multi-Head Attention Module to learn more important weights.

Both the lower complexity and rolling window sequences aim to make the first layer attention as expressive and useful to the dynamic global predictor as possible. However, this sacrifices the

stability of learning that the original model had. We are now training a transformer for attention harvesting, but its loss is still tuning the transformer to predict the correct takenness of the next branch in a sequence. We see in Figure 7.3 that the lower complexity of our rolling window transformer has a worse performance predicting branches and less stable training. The rolling window nature of the input pre-processing means branches are seen several times meaning fewer epochs can be used in the future. Although the training of our new architecture is less stable, the lower complexity does not hurt accuracy significantly. In Figure 7.4 we see that the new model produces significantly more expressive attention matrices. This was the goal of the new model and demonstrates we have successfully designed a transformer architecture tuned for attention matrix harvesting. We believe further model simplification and alternative attention mechanisms could yield even more useful attention matrices produced in Layer 0.



(a) Original Transformer validation accuracy per epoch

(b) Rolling Window Transformer validation accuracy per epoch

Figure 7.3: Comparison of accuracy on the validation set per epoch between the Original Transformer and the Rolling Window Transformer. The Rolling Window Transformer has much less stable learning and often does not benefit from many epochs due to its lower complexity.

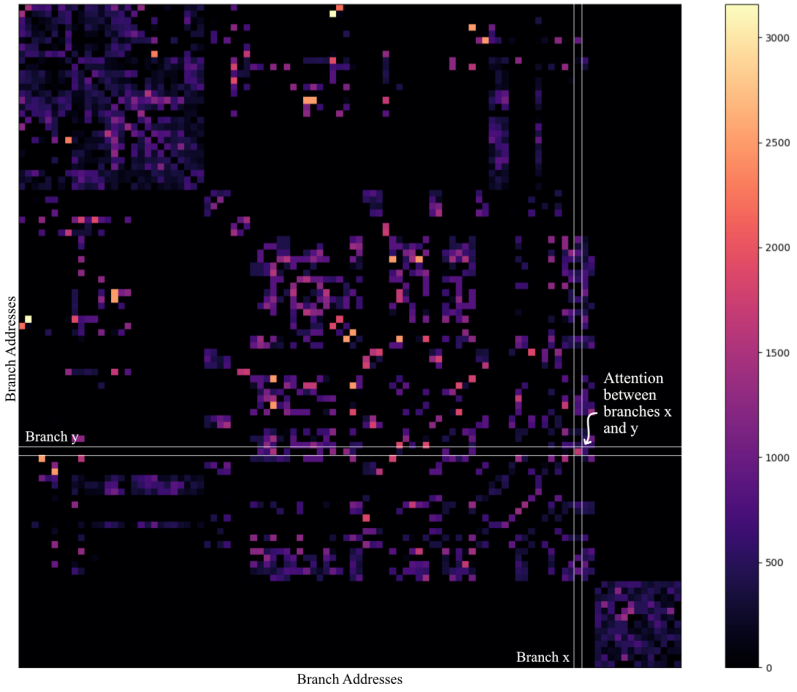


Figure 7.4: Cropped global attention map using a rolling window transformer in the SPEC2006 gobmk. Cropped into identical branch addresses as Figure 6.2. We see a more expressive global attention map.

We trained this new transformer architecture on all training set traces, extracted the attention map from them, and generated colour labels using the fast greedy community detection algorithm as before.

7.3 Results

We see that there is a performance improvement over the original transformer-based labels using our rolling window transformer. However, on average there is a decrease in predictor performance due to astar. We see the sjeng and gobmk benchmarks be greatly improved by the addition of the rolling-window transformer-based labels.

Benchmark	Default Tournament Misses-per-Kilo-Instruction	Augmented Tournament Misses-per-Kilo-Instruction *Global Rolling Window Transformer-based Labels*	Percentage Improvement
sjeng	14.09	13.67	2.93
mcf	20.63	20.47	0.76
hmmer	1.71	1.71	0.16
h264ref	1.57	1.55	0.91
gobmk	19.68	19.14	2.76
bzip2	6.59	6.58	0.08
perlbench	6.04	6.06	-0.35
xalancbmk	4.22	4.28	-1.59
astar	37.25	39.54	-6.15
gcc	7.49	7.48	0.08
omnetpp	5.81	5.91	-1.84
Average	11.37	11.49	-1.08

Figure 7.5: Performance difference between the original tournament branch predictor and a colour augmented tournament branch predictor fed with global colour labels derived from the learnt attention of a rolling window transformer. Lower Misses-per-Kilo-Instruction is better.

The rolling window transformer-based labels can improve on the state-of-the-art in several programs in branch prediction performance, an area that is not only incredibly difficult to improve on, but also paramount to the performance of modern computing. We believe these results show that the natural language processing approach to generating colours for our augmented tournament branch predictor is a powerful method and further refinements would yield even more favourable results.

7.3.1 Dual Colouring Scheme

As before we tested both the local takenness-based labels and global rolling window transformer-based labels simultaneously. We see that they complement each other and in some benchmarks produce over 4% improvement. However, we note that in the average case, we see performance drop due to star. The improvements in some benchmarks indicate that both the augmented tournament architecture and clustering algorithms we use can be powerful at improving branch prediction in the right conditions.

Benchmark	Default Tournament Misses-per-Kilo-Instruction	Augmented Tournament Misses-per-Kilo-Instruction *Global Rolling Window Transformer-based and Local Takenness-based Labels*	Percentage Improvement
sjeng	14.09	13.40	4.88
mcf	20.63	20.40	1.11
hmmer	1.71	1.71	0.16
h264ref	1.57	1.53	2.01
gobmk	19.68	18.78	4.60
bzip2	6.59	6.58	0.09
perlbench	6.04	5.81	3.79
xalancbmk	4.22	4.29	-1.77
astar	37.25	39.54	-6.16
gcc	7.49	7.46	0.36
omnetpp	5.81	5.73	1.42
Average	11.37	11.38	-0.13

Figure 7.6: Performance difference between the original tournament branch predictor and a colour augmented tournament branch predictor fed with takenness local colour labels and rolling window transformer derived global colour labels simultaneously. Lower Misses-per-Kilo-Instruction is better.

7.4 Summary

In this chapter, we improved our transformer-based clustering algorithm using a rolling window approach.

Chapter 8

Evaluation

In this chapter, we will evaluate the performance of our custom augmented branch predictor in the Gem5 simulator when fed with local and global colour labels generated by the clustering algorithms we developed.

8.1 Performance of our Augmented Tournament Predictor

In this thesis, we developed a novel branch predictor, a colour augmented version of the tournament predictor, which not only learns from the dynamic branch history but can also be fed with local and global branch colour labels as an extra source of information. We have seen that modern state-of-the-art branch prediction is very powerful and approaching the theoretical limit with the current information supplied to them [1], therefore we sought to expand the way we think about branch prediction by supplying the current state-of-the-art predictors with extra statically derived information in the form of colour labels.

We designed several clustering algorithms to generate these colour labels, which included expanding on the work of Sburlan's hypothetical transformer branch predictor [3] and producing a statistical model of branch prediction. The two most powerful clustering algorithms we produced are a Local Takenness-based scheme and a Global Rolling Window Transformer-based scheme. In Figure 8.1 we take the best clustering schemes per benchmark in the Spec2006 suite and compare our novel branch predictor with the state-of-the-art comparisons presented by Sburlan. We see tangible improvements in prediction accuracy in every benchmark versus TAGE-SC-L 64KB while maintaining a smaller memory footprint. We also see significant improvements versus the hypothetical transformer model while being a more reasonably implementable predictor architecture.

In Figure 8.2, we present the performance improvement we produce over the default tournament predictor. In a field where state-of-the-art prediction accuracy is excellent, we were able to improve it further in many benchmarks. These results show the merit of the augmented tournament architecture and suggest that improvements to the clustering algorithms could produce even bigger improvements with the same branch predictor architecture. We note, however, that this is the best case for every benchmark meaning a reliable general-purpose clustering scheme to generate global colour labels would need further research. However, we believe our local Takenness-based labels are the optimal use of 1 bit of colour information and manage to improve prediction accuracy reliably and in the same memory footprint as the local predictor in the default state-of-the-art tournament.

Overall, we believe that the results achieved by our augmented tournament architecture and clustering algorithms working in tandem are very promising and could warrant further research into optimising clustering algorithms to supply the most useful information. Our novel branch predictor aimed to broaden the information available to branch predictors and managed to improve the performance of state-of-the-art predictors in most benchmarks in the SPEC2006 suite in a field where improvements are few and far between. Furthermore, our proposed augmented tournament architecture can provide these performance benefits while being both implementable and no more memory intensive than its competitors and its flexible design opens the door to future research into experiments for increasingly better colour label assignments and prediction power.

Benchmark	TAGE-SC-L 64KB Accuracy (%) [3]	Transformer Accuracy (%) [3]	Augmented Tournament Best Case Accuracy (%)	Best Case Labelling Scheme		Delta Accuracy (%)	
				Local	Global	to TAGE -SC-L	to Trans- former
sjeng	91.6	91.1	94.7	Ta	RWT	3.1	3.6
mcf	90.4	94.5	97.0	Ta	RWT	6.6	2.5
hmmer	94.9	90.4	98.0	Ta	RWT	3.1	7.6
h264ref	95.6	97.7	98.5	Ta	RWT	2.9	0.8
gobmk	92.1	94.1	93.5	Ta	RWT	1.4	-0.6
bzip2	94.1	94.9	97.0	Ta	RWT	2.9	2.1
perlbench	94.1	83.0	97.6	Ta	-	3.5	14.6
xalancbmk	98.3	87.8	98.5	Ta	-	0.2	10.7
astar	84.3	90.9	92.6	Ta	-	8.3	1.7
gcc	96.8	91.3	97.7	Ta	-	0.9	6.4
omnetpp	97.7	82.8	97.7	Ta	-	0.0	14.9
Average	93.6	90.8	96.6			3.0	5.8

Key for Best Case Labelling Scheme	
Ta	1 bit colour of takenness-based labels.
RWT	2 bits colour of Rolling Window Transformer-based labels.
-	0 bits colour. Default predictor.

Figure 8.1: Performance comparison between TAGE-SC-L 64KB, the pure Transformer model [3], and our augmented tournament predictor fed with best case labels. Higher Accuracy is better. Accuracy calculated as = 100 * Misspredicts/Total Lookups.

Benchmark	Default Tournament Accuracy (%) [3]	Augmented Tournament Best Case Accuracy (%)	Best Case Labelling Scheme		Delta Accuracy (%)
			Local	Global	
sjeng	94.58	94.72	Ta	RWT	0.15
mcf	97.02	97.04	Ta	RWT	0.02
hmmer	97.96	97.96	Ta	RWT	0.00
h264ref	98.52	98.53	Ta	RWT	0.02
gobmk	93.32	93.54	Ta	RWT	0.21
bzip2	96.96	96.96	Ta	RWT	0.00
perlbench	97.52	97.65	Ta	-	0.13
xalancbmk	98.50	98.49	Ta	-	-0.01
astar	92.63	92.63	Ta	-	0.00
gcc	97.66	97.69	Ta	-	0.03
omnetpp	97.63	97.68	Ta	-	0.05
Average	96.57	96.63			0.05

Figure 8.2: Performance comparison between the Default Tournament model [3] and our Augmented Tournament Predictor fed with best case labels. Higher Accuracy is better. Accuracy calculated as 100 × Misspredicts/Total Lookups. Best Case Labelling Scheme Key in Figure 8.1.

Chapter 9

Conclusion and Future Work

9.1 Summary of Results

This thesis aimed to develop a branch predictor that learns dynamically but utilises static information to incorporate more context into its prediction. We believe we have been successful in this goal, as this work has produced a novel augmented tournament branch predictor architecture and several clustering algorithms that provide static context sources for it in the form of colour labels. The most important contributions of this work are:

- Design and simulation of a novel branch predictor (augmented tournament branch predictor) that can be fed with separate colour labels for both the local and global sub-predictors to widen the context it uses to make predictions. Our novel branch predictor can make predictions based on the dynamic history, like current state-of-the-art predictors, but also learn from a branch's colour label which carries extra context information.
- A takenness-based clustering algorithm to generate colours for the local predictor in the augmented tournament branch predictor. Compared to the default state-of-the-art tournament predictor, we found an average improvement of 1.27% Misses-per-Kilo-Instruction across the SPEC2006 CPU benchmark suite, with several benchmarks seeing improvements above 2.5% and perlbench seeing improvements of 5.5%.
- Mathematical proof on the utility of the takenness-based clustering algorithm improves predictions in a constrained memory footprint for a general program.
- A rolling window transformer-based clustering algorithm to generate colours for the global predictor in the augmented tournament branch predictor. When combined with the local takenness-based labels, compared to the default state-of-the-art tournament predictor, we found several benchmarks seeing improvements over 3% Misses-per-Kilo-Instruction. Compared to the state-of-the-art TAGE-SC-L 64KB predictor, our augmented tournament branch predictor fed with optimal labels achieves a 3.0% average improvement in prediction accuracy on the SPEC2006 CPU benchmark suite, all while using a smaller memory footprint.
- A study into how to modify a transformer architecture to make its learnt attentions as expressive and useful to the clustering algorithm as possible while not modifying its loss function.

9.2 Limitations

While our results are promising we aim to make this work as useful to the wider academic community as possible. In this section, we aim to provide the known limitations of this work, such that further research extending our findings can address them or take them into account. The limitations of our work are:

- The traces of programs are formed using SimPoints to allow efficient and fast simulation of programs. However, they only approximate full program execution which means they do not have 100% coverage of the code.

- While the rolling window transformer-based colour labels did improve prediction accuracy in several SPEC2006 CPU benchmarks, the large performance drop in a few meant the average over the suite dropped slightly. This suggests the model needs modification to produce a more reliable set of colours that help all programs.

- The rolling window transformer model introduces several new parameters such as the window size and step progression. The remainder of the architecture shape was tuned by hand using intuition on how the attention matrices would react. A Bayesian Optimisation or alternative hyper-parameter search method would be a more optimal way to tune these.

- Evaluating results on a predefined benchmark suite runs the risk of steering research towards optimising the benchmarks rather than actual workloads used by people in practice.

9.3 Future Work

In this section, we will detail several possible ideas and research directions we believe are avenues for future work with high potential for further improving branch prediction accuracy.

- **A Statistical Model of the Augmented Global Predictor and its Labels:** In this thesis we present an analysis of the impact of takenness-based labels based on a statistical model of the local predictor. This allowed us to produce a clustering algorithm that confidently provides beneficial context to the local predictor. The development of a model for the global predictor would allow a concrete means to analyse and compare global labelling schemes, which would pave the road for discovery of the reliable and optimal clustering algorithms to inject extra context into our augmented tournament branch predictor.

- **Experimentation into Discovery of New Context Sources:** By creating a branch predictor independent of the colouring scheme, we provide the framework for future work to colour branches in unique and powerful ways. New ways of colouring branches can inject any number of contexts and data sources into our novel branch predictor. We have created a flexible predictor architecture that can learn from information never before used in branch prediction and opens up a new way of looking at improving branch prediction. Future work to generate colours based on never-before-used information sources, for example using the trace-time Return Address Stack (RAS) state, would be immensely promising.

- **Further Modifications of the Transformer Architecture:** We developed the Rolling Window Transformer in a bid to make more expressive attention matrices. However, we believe there are several ways in which we can use the latest Natural Language Processing techniques to improve the architecture further:

  - **New Loss Function:** Our transformer uses a binary cross entropy loss function to help the transformer predict what the outcome of the next token in the sequence is. However, we do not use these predictions directly, so an alternative loss function directed at assigning colours may be an alternative angle to approach the clustering problem. A transformer that directly assigns colours to branches would be a powerful way to add context to a dynamic predictor if a training dataset can be created or sourced. Furthermore, this approach would open up new avenues for the architecture of the transformer:

    * Deeper architecture as attention matrix expressiveness is no longer a goal.
    * Utilising a pre-trained transformer model and fine-tuning it for a specific program offers significant advantages, eliminating the need for tokens to map directly to branch addresses and allowing the model to consider both branch addresses and disassembly. This approach reduces compile-time training and leverages the extensive knowledge from training on large datasets, thus enhancing efficiency and accuracy. Pre-trained transformers already possess learned representations of code patterns and structures, which, when fine-tuned, adapt to the specific characteristics of the target program, improving prediction accuracy. Training on multiple programs allows the model to develop a generalised understanding of code behaviours,

making it robust and versatile. By integrating both branch addresses and disassembly instructions, the model gains a richer context, enabling more informed and reliable predictions.

  - **Sparse Attention Mechanism:** Research by Child et al. [33] for OpenAI developed a new attention mechanism for long sparse inputs. The attention mechanism breaks the input sequence into chunks and assigns attention to these larger chunks rather than individual tokens. This is a promising new architecture to generate more expressive global attention maps as it would allow large sequence lengths while discouraging smoothness across large swaths of the matrix. The principle of attending to chunks on tokens has a natural symmetry with the idea of program phases and we believe it could be a fruitful avenue for future research. Further reduction in the depth of the encoder and decoder would also help complement a sparse attention model and improve the expressiveness of the global attention map.

- **Experimentation in Resource-Constrained Use Cases:** We evaluated the improvements of our novel branch predictor and clustering algorithms assuming a large state-of-the-art memory footprint for all our branch predictors. However, not all computation is done in processors that have these large resource allocations for branch predictors. Embedded systems such as microcontrollers and memory-constrained ASICs, often operate under severe resource limitations. In these systems, every bit of memory and processing power is precious, and traditional branch predictors with large memory footprints may not be feasible or practical. As such, they often have far smaller dynamic histories available to them and perform with lower accuracy. By supplementing one bit of colour information in a memory-constrained system, it could combat limited runtime information with powerful static context. Improvements from our colour augmentation in prediction accuracy in these smaller memory predictors would likely be significantly larger than what we see in our evaluation.

- **Reducing Memory Footprint of the Augmented Global Predictor:** Our augmented local predictor maintains the same memory footprint as its unaugmented counterpart as we are confident in our ability to feed it useful context, however, we noted that the memory requirements of our augmented global predictor are increased limit damage from poor clustering algorithms. However, if future research produces reliably useful global colours then the deeper tournament structure can be removed. This would make the space requirements of the augmented predictor identical to the McFarling predictor [11].

- **Recognise Program Branch Predictor Phases using BBVs and SimPoint:** BBVs and SimPoints can be used to define behaviour phases in a program which we can use to assign colours to branches. While there is no guarantee these behaviour patterns align with the branch predictor phases, we believe they may be similar. Furthermore, we could change the metrics SimPoint looks at to assign behaviour phases such that it focuses more on attributes relevant to branch prediction.

9.4 Ethical Considerations

Branch prediction is crucial in modern processors, however, it can be exploited through branch prediction attacks. Spectre and Meltdown attacks have highlighted vulnerabilities in branch prediction, affecting high-performance processors. Despite efforts to mitigate these risks, no comprehensive solution has emerged. We believe our research does not exacerbate these issues, but the incorporation of program context through colour labels offers an extra avenue of attack for malicious actors. We believe the closed compiler time generation of the colour labels and the fact our novel branch predictor does not commit speculative instructions and can benefit from the same mitigations for side-channel attacks currently in use. However, further detailed research into uncovering any new channels of attack is crucial before deploying our novel branch predictor into practice.

The world of computer architecture research is often obfuscated and often private due to the industry's competitive nature. Commercial interests often restrict information sharing and hinder academic progress. To foster transparency and encourage further research expanding on ours, our work is open-source. This includes all ML models, branch predictor simulation code and

clustering algorithms. Our evaluation utilised the SPEC2006 CPU benchmark suite and the Gem5 microarchitecture simulator, following licensing agreements for their use.

Our research used extensive computing power to produce results. We estimate 1500 hours of NVIDIA Quadro RTX 6000 runtime and 1000 hours of Intel Xeon CPU E5-2640 runtime were required during the research and data collection phases of this project. This amounts to a non-negligible carbon footprint. Based on carbon efficiency of UK power grids and power draw of hardware components, total emissions are estimated to be roughly 100 kg CO_2 eq [34]. However, we believe our research will be able to speed up the future of computing. Better branch prediction will allow more computations to be done in the same power envelope and reduce the carbon emissions of future silicon.

9.5 Reflections

We believe our research was successful in fulfilling the goals we set out to, however, during this project, we made some wrong turns that cost time and pointed us in the wrong direction. The main wrong turn we took was designing a custom attention mechanism for the transformer. We attempted to implement a combination attention mechanism with rolling window aspects of a Longformer while also being sparse. This meant rewriting the entire transformer architecture from scratch. However, after dedicating a lot of project time to it, initial testing saw unexplained poor learning which we believe to be caused by bugs in implementation. Ultimately we abandoned it and switched to the rolling window transformer architecture we present in this thesis, however, it provided us with great insight into areas for future research.

Appendix A

Analysis of Local Takenness-Based Colour Utility

To show why takenness-based colours are useful to a local branch predictor in the vast majority of programs, we define a hypothetical statistical model of a general-purpose program with some assumptions. This work is referenced in chapter 5.1.1.

A.1 *Define Local Predictor*

We now define the model for a local branch predictor mechanism, using the law of large numbers (LLN) [35] and assuming branch behaviour is independent. This of course obfuscates the nuances of individual branch behaviour but allows us to study a local branch predictor in the worst-case scenario. If colour labels can improve the worst case in the same memory footprint, it bounds the minimum performance of the predictor, which is very valuable in improving overall performance.

For simplicity, let us assume we collected all the branches in the branches and reassigned them pseudo program counters, such that they are consecutive. Let PC be this pseudo program counter, which we model as a random variable.

Let there be N branches in the program. The PC at time t, PC_t, is determined based on the previous PC, PC_{t-1}, and has a certain probability p of being completely random as the branch is taken. We cannot make any further assumptions about the next PC under this taken case. Formally, we can express this as:

$$\text{PC}_t = f(\text{PC}_{t-1}) = \begin{cases} \text{PC}_{t-1} + 1 & \text{with probability } 1-p \\ \text{Uniform}(0, N-1) & \text{with probability } p \end{cases}$$

where f is a function that determines the next PC based on the current PC, and $\text{Uniform}(0, N-1)$ represents a random PC drawn uniformly from the address space $[0, N-1]$.

Next, we define the function, g, that takes the least significant $log_2(M)$ bits from PC_t to hash to the local history table and return the index local history table required, I. Given that $N \gg M$, we define a surjective function g as follows:

$$g : \{0, 1, 2, \ldots, N-1\} \to \{0, 1, 2, \ldots, M-1\}$$

The function g and the local history table index, I, is defined by:

$$I = g(PC_t) = PC_t \bmod M \text{ ; where mod denotes the modulo operation.}$$

The function h then returns the local history value for index h at time t, LH_t^I. The time t is particular to the local history table and independent of the PC, but we will use t in both places

for ease of notation. We also can use the average branch takenness of this program, p. Given the local history LH^I_{t-1}, the function h is defined by:

$$h : \{LH^I_{t-1}\} \to \{LH^I_t\}$$

where:

$$LH^I_t = \begin{cases} \left\lfloor \frac{LH^I_{t-1}}{2} \right\rfloor \times 2 + 0 & \text{with probability } 1-p \\ \left\lfloor \frac{LH^I_{t-1}}{2} \right\rfloor \times 2 + 1 & \text{with probability } p \end{cases}$$

Here, $\lfloor \cdot \rfloor$ denotes the floor function, which effectively drops the most significant bit of LH^I_{t-1}, and the term $\times 2$ shifts the remaining bits left by one position to make space for the new least significant bit.

We then define the counter entry, C, corresponding with LH^I_{t-1} to be. Note we use LH^I_{t-1}, not LH^I_t as the local history is not updated during look-up.

$$Prediction = C_{LH^I_{t-1}}$$

The update rule for C, which occurs after the prediction is returned, is:

$$C_{LH^I_{t-1}} = \begin{cases} \max(C_{LH^I_{t-1}} - 1, 0) & \text{with probability } 1-p \\ \min(C_{LH^I_{t-1}} + 1, \text{MAX}) & \text{with probability } p \end{cases}$$

Here, min and max functions ensure that the saturating counter remains within the range $[0, \text{MAX}]$. We have now defined the entire probability transition function space from PC to prediction with a local branch predictor.

A.2 *Local Predictor Behaviour without Takenness-based Labels*

We will now use the local branch predictor definition to study its default behaviour concerning the 2 problem cases we outlined before. First, let us assume this program runs for long enough that the function f reaches its steady state. This is a reasonable assumption for most programs. We then define the state transitions for the function f:

- *Initial State*: Assume PC_0 is initially uniformly distributed over the range $[0, N-1]$.
- *Transition*: At each step, PC_t is equal to PC_{t-1} with probability $1-p$, and is a uniformly random value from $[0, N-1]$ with probability p.

  Let's denote $P(\text{PC}_t = i)$ as the probability that PC_t is equal to some specific value i.

  Given the transition probabilities:

$$P(\text{PC}_t = i) = (1-p) \cdot P(\text{PC}_{t-1} = i) + \frac{p}{N}$$

To find the steady-state distribution, we assume that the distribution of PC_t reaches a point where it does not change from one step to the next. Let this steady-state probability be $P(\text{PC} = i) = q$. In the steady state, we have:

$$q = (1-p) \cdot q + \frac{p}{N}$$

Solving for q:

$$pq = \frac{p}{N}$$

$$q = \frac{1}{N}$$

This implies that, in the steady state, each PC value is equally likely, with probability $\frac{1}{N}$. So, with these assumptions of running to steady state, **PC is distributed uniformly** over $\{0, 1, 2, \ldots, N-1\}$ where $N \gg M$.

To show that I is also uniformly distributed, we need to show that $P(I = k)$ is the same for all $k \in \{0, 1, \ldots, M-1\}$.

Since PC is uniform, $P(\text{PC} = i) = \frac{1}{N}$ for all $i \in \{0, 1, \ldots, N-1\}$.
For a given $k \in \{0, 1, \ldots, M-1\}$, the values of PC that satisfy PC mod $M = k$ are:

$$k, k+M, k+2M, \ldots, k + \left(\left\lfloor \frac{N-1-k}{M} \right\rfloor \right) M$$

The number of such values is $\left\lfloor \frac{N-1-k}{M} \right\rfloor + 1 \approx \frac{N}{M}$.
Thus, the probability $P(I = k)$ is:

$$P(I = k) = \sum_{j=0}^{\left\lfloor \frac{N-1-k}{M} \right\rfloor} P(\text{PC} = k + jM) = \left(\left\lfloor \frac{N-1-k}{M} \right\rfloor + 1 \right) \frac{1}{N} \approx \frac{N/M}{N} = \frac{1}{M}$$

Since this is true for all $k \in \{0, 1, \ldots, M-1\}$, I **is uniformly distributed** over $\{0, 1, \ldots, M-1\}$.

With this, we can consider the 2 cases of concern:

- **1 Two branches that alias to the same history register are being seen by the program at the same time:**

  Let the length of local history be LH_{len}. Let us assume the local history and counters have been fully primed for PC_i as it has been the only branch they have seen and they have seen it sufficient times. Given we have just seen a branch PC_i, it has primed $LH^{\text{PC}_i mod M}$ and its corresponding counters.

  We then sample from the uniform PC distribution and local history index distribution as proven before. For two (or more) branches to alias into the same history register and be seen by the program at the same time, we must see PC_i as one of the next LH_{len} PCs that index into $LH^{\text{PC}_i mod M}$.

  Given that the probability of getting PC_i as the immediate next branch is $\frac{M}{N}$, we can use the cumulative distribution function of the geometric distribution to determine the probability P of getting into this simultaneous aliasing situation is:

  $$P(\text{Simultaneous Aliasing}) = 1 - \left(1 - \frac{M}{N}\right)^{\text{LH}_{\text{len}}} \tag{A.1}$$

  Although we see different aliased branches before seeing PC_i, how much the aliasing changes the local predictor state is a function of both the number of times a branch that is not PC_i is seen and how different these branches behave to PC_i. We can estimate how much damage aliasing will do by finding the probability that the branching pattern of PC_i matches an aliased branch PC_j. Both PC_i and PC_j must have probability of being taken p_A and p_B respectively drawn from a normal distribution with mean p (the branch takenness of the entire program). So, define p_A and p_B to be probabilities drawn independently from $\mathcal{N}(p, \sigma^2)$, where the standard deviation is dependent on the program. The probability density function of $p \sim \mathcal{N}(p, \sigma^2)$ is:

  $$f_{\mathcal{N}}(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right)$$

  The probability of a sequence of N branches match between PC_i and PC_j can then be given by:

  $$P(\text{sequence match}) = \int_0^1 \int_0^1 \left(\sum_{k=0}^{N} \binom{N}{k}^2 p_A^k p_B^k (1-p_A)^{N-k} (1-p_B)^{N-k} \right) f_{p_A}(p_A) f_{p_B}(p_B)\, dp_A\, dp_B$$

where f_{p_A} and f_{p_B} are the PDFs for p_A and p_B. We know probability distributions that p_A and p_B are drawn from:

$$P(\text{sequence match}) = \int_0^1 \int_0^1 \left(\sum_{k=0}^{N} \binom{N}{k}^2 p_A^k p_B^k (1-p_A)^{N-k} (1-p_B)^{N-k} \right) f_{\mathcal{N}}(p_A) f_{\mathcal{N}}(p_B)\, dp_A\, dp_B \tag{A.2}$$

- **2 Two branches that alias to the same history register are being seen by the program at different times:**

  Given we have the predictor primed with the branch pattern of PC_i, we want to know the probability that the next branch, PC_j, that inherits its counter and history is predicted correctly.

  Let PC_i and PC_j have probability of being taken p_A and p_B respectively as before. Also, as before define p_A and p_B to be probabilities drawn independently from $\mathcal{N}(p, \sigma^2)$, where the standard deviation is dependent on the program. The local predictor is primed to PC_i, so will return taken with p_A. When it encounters PC_j it will predict taken with probability p_A. The probability this is correct is:

$$P(PC_j \text{ predicted correctly}) = 2p_A p_B - p_A - p_B + 1 \tag{A.3}$$

  This is maximised, when $p_A = 0$, $p_B = 0$ or $p_A = 1$, $p_B = 1$. For a fixed p_A, it is maximised when $p_A = p_B$.

A.3 *Local Predictor Behaviour with Takenness-based Labels*

We will now do a similar study on the 2 problem cases we outlined before, but with a model modified with colour labels.

With takenness labels, we want a class of branches that are 'more often taken' and a class that are 'more often not taken'. For this analysis lets assume unseen branches are automatically put into the 'more often not taken' class. Our implementation optimises this choice further, but can be proved in an identical way.

We know about the program is on average branches are taken with probability p, and not taken with probability $1 - p$. We also have shown that the PCs are uniformly distributed when the program runs for long enough. Let all PCs ($PC_0 ... PC_{N-1}$) be an RV that draws from a binomial distribution. Each branch outcome is a Bernoulli trial, and we define X_i as the number of "taken" outcomes in n outcomes of PC_i:

$$X_i \sim \text{Binomial}(n, p)$$

We seek the probability that there are more "taken" outcomes than "not taken" outcomes, i.e.,

$$P\left(X_i \geq \frac{n}{2}\right)$$

Case 1: n is even, so let $n = 2k$. We need $X_i \geq k$:

$$P(X_i \geq k) = 1 - P(X_i < k) = 1 - F(k; n, p)$$

Case 2: n is odd, so let $n = 2k + 1$. We need $X_i \geq k + 0.5$, which is the same as $X_i > k$:

$$P(X_i > k) = 1 - P(X_i \leq k) = 1 - F(k; n, p)$$

Combining both cases:

$$P\left(X_i \geq \frac{n}{2}\right) = 1 - F\left(\left\lfloor \frac{n}{2} \right\rfloor; n, p\right) = 1 - \sum_{j=0}^{\left\lfloor \frac{n}{2} \right\rfloor} \binom{n}{j} p^j (1-p)^{n-j}$$

$$P\left(X_i < \frac{n}{2}\right) = F\left(\left\lfloor \frac{n}{2} \right\rfloor; n, p\right) = \sum_{j=0}^{\left\lfloor \frac{n}{2} \right\rfloor} \binom{n}{j} p^j (1-p)^{n-j}$$

where $F(x; n, p)$ is the cumulative distribution function (CDF) of the Binomial distribution, and $\binom{n}{i}$ is the binomial coefficient.

Let us assume $\theta\%$ of branches are seen in the training set, and the training set is representative of the program. We have $\frac{\theta}{100}N$ branches in our training set, so: $\frac{\theta}{100}N(1 - F\left(\left\lfloor \frac{n}{2} \right\rfloor; n, p\right))$ branches are in the 'more often taken' class and have probability of being taken ≥ 0.5. Similarly, $\frac{\theta}{100}N(F\left(\left\lfloor \frac{n}{2} \right\rfloor; n, p\right))$ branches are in the 'more often taken' class and have probability of being taken < 0.5.

However, at runtime the $\frac{100-\theta}{100}N$ unseen branches are assigned to the 'more often not taken' class. If $p \leq 0.5$ then the new average probability of branches in the 'more often not taken' class being taken is < 0.5, independent of θ. If $p > 0.5$ then the unseen branches pull the average towards p. Let x be the initial training set probability of branches in the 'more often not taken' class being taken:

$$P(taken|\text{'more often not taken' class}) = \frac{\left(\frac{\theta}{100}N(F\left(\left\lfloor \frac{n}{2} \right\rfloor; n, p\right))\right)x + \left(\frac{100-\theta}{100}N\right)p}{\frac{\theta}{100}N(F\left(\left\lfloor \frac{n}{2} \right\rfloor; n, p\right)) + \frac{100-\theta}{100}N}$$

$x < 0.5$. So:

$$P(taken|\text{'more often not taken' class}) < p$$

Now we have two classes of branches. In the worst case, branches in 'more often not taken' class have probability of being taken $<$ p, and branches in 'more often taken' class have probability of being taken > 0.5.

The PC is supplied uniformly as shown previously. However, we now use only $log_2(M) - 1$ bits from PC_t to hash to the local history table. Given that $N \gg M/2$, we define a new surjective function g' as follows:

$$g' : \{0, 1, 2, \ldots, N-1\} \rightarrow \{0, 1, 2, \ldots, \frac{M}{2} - 1\}$$

The function g' and the local history table index, I', is defined by:

$$I' = g'(PC_t) = PC_t \bmod \frac{M}{2}$$

Similarly to our previous proof, I' is uniformly distributed over $\{0, 1, \ldots, \frac{M}{2} - 1\}$.

I' has the 'more often taken' label prepended to it. We model this as two separate local history tables, $LHTaken$ and $LHNotTaken$ as opposed to the previous LH:

– *Branch is in 'more often taken' class:*

$$LHTaken_t^{I'} = \begin{cases} \left\lfloor \frac{LHTaken_{t-1}^{I'}}{2} \right\rfloor \times 2 + 0 & \text{with probability } (1 - p_t) < 0.5 \\ \left\lfloor \frac{LHTaken_{t-1}^{I'}}{2} \right\rfloor \times 2 + 1 & \text{with probability } p_t \geq 0.5 \end{cases}$$

– *Branch is in 'more often not taken' class:*

$$LHNotTaken_t^{I'} = \begin{cases} \left\lfloor \frac{LHNotTaken_{t-1}^{I'}}{2} \right\rfloor \times 2 + 0 & \text{with probability } (1 - p_{nt}) \geq p \\ \left\lfloor \frac{LHNotTaken_{t-1}^{I'}}{2} \right\rfloor \times 2 + 1 & \text{with probability } p_{nt} < p \end{cases}$$

With this, we can consider the 2 cases of concern:

- **1 Two branches that alias to the same history register are being seen by the program at the same time:**

  We will now derive the same metrics we derived for the case with uncoloured branches, under identical situations.

  By using one fewer bit of the PC, we have a higher likelihood of aliasing. We must also consider the two classes separately:

$$\begin{aligned}&P(\text{Simultaneous Aliasing})\\&= P(\text{Simultaneous Aliasing} \mid \text{PC}_i \in \text{'most often taken' class})\\&\quad + P(\text{Simultaneous Aliasing} \mid \text{PC}_i \in \text{'most often not taken' class})\end{aligned}$$

  We have shown before there are $\frac{\theta}{100}N(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right))$ branches are in the 'more often taken' class and they have probability of being taken ≥ 0.5. Likewise, there are $N - \frac{\theta}{100}N(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right))$ branches are in the 'more often not taken' class and they have probability of being taken $<$ p. Therefore we have:

$$\begin{aligned}&P(\text{Simultaneous Aliasing} \mid \text{PC}_i \in \text{'most often taken' class})\\&= 1 - \left(1 - \frac{M}{2\frac{\theta}{100}N(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right))}\right)^{\text{LH}_{\text{len}}}\end{aligned}$$

$$\begin{aligned}&P(\text{Simultaneous Aliasing} \mid \text{PC}_i \in \text{'most often not taken' class})\\&= 1 - \left(1 - \frac{M}{2(N - \frac{\theta}{100}N(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right)))}\right)^{\text{LH}_{\text{len}}}\end{aligned}$$

  Then for P(Simultaneous Aliasing), we simply substitute the expressions:

$$\begin{aligned}&P(\text{Simultaneous Aliasing})\\&= 1 - \left(1 - \frac{M}{2\frac{\theta}{100}N(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right))}\right)^{\text{LH}_{\text{len}}}\\&\quad + 1 - \left(1 - \frac{M}{2(N - \frac{\theta}{100}N(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right)))}\right)^{\text{LH}_{\text{len}}}\end{aligned}$$

  Comparing this to equation A.1, P(Simultaneous Aliasing) are identical in the two cases only if $\frac{\theta}{100}(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right)) = 0.5$:

$$\lim_{\substack{\theta \to 100 \\ n \to \infty}} \frac{\theta}{100}(1 - F\left(\left\lfloor\frac{n}{2}\right\rfloor; n, p\right)) = 0.5$$

  This expression indicates that as θ approaches 100 and n approaches infinity, the given function approaches 0.5 - our ideal situation in which we can use fewer bits of the PC to index the local history table with no increase in aliases. These limits signify we approach the ideal situation as the traces cover more possible branches and the traces see any given branch as many times as possible. So ***the extra alias cost of our colour labels maintaining the same memory footprint is a function of how good the training set traces are***.

  We can also find the counterpart to equation A.2. Our new branch takenness probabilities are bounded, which modifies the bounds on the integrals in equation A.2. For the situation our branches that alias into each other are in the 'more often not taken' class:

Branch outcomes are drawn from a new 'truncated' normal distribution. The PDF for p_A and p_B, given $p_A, p_B < p$, is:

$$g(x) = \frac{\frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right)}{0.5} = \frac{2}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-p)^2}{2\sigma^2}\right)$$

This ensures the integral of the truncated PDF over $[-\infty, p)$ is 1. Using the truncated PDF $g(x)$:

$$P(\text{sequence match}) = \int_0^p \int_0^p \sum_{k=0}^{N} \binom{N}{k}^2 p_A^k p_B^k (1-p_A)^{N-k} (1-p_B)^{N-k} g(p_A) g(p_B)\, dp_A\, dp_B$$

Given the truncation (less than p), we outcomes of the aliased branches, p_A and p_B will have smaller variation, making both coins more likely to show heads. Therefore, the sequence match probability for two branches in the 'more often not taken' class is expected to be higher compared to the uncoloured case seen in equation A.2.

Similarly for the 'more often taken' class, the normal distribution $\mathcal{N}(p, \sigma^2)$ truncated at 0.5 can be written with a scaled PDF to ensure its integral is 1. Let this new PDF be $g'(x)$. Using $g'(x)$:

$$P(\text{sequence match}) = \int_{0.5}^1 \int_{0.5}^1 \sum_{k=0}^{N} \binom{N}{k}^2 p_A^k p_B^k (1-p_A)^{N-k} (1-p_B)^{N-k} g'(p_A) g'(p_B)\, dp_A\, dp_B$$

Given the truncation (greater or equal to 0.5), we expect the p_A and p_B values to be larger, making both branches more likely to be taken. Therefore, the sequence match probability is expected to be higher compared to the uncoloured case in equation A.2 because there is less variation and hence a lower likelihood of divergence in sequences.

So, we have proven in both classes ***there is an improvement in the probability that sequence match, and so a decrease in branch interference***. So using colours to assign branches into these classes will, on average, be beneficial for predictor accuracy, given that the dataset can divide branches close to a 50/50 split.

- **2 Two branches that alias to the same history register are being seen by the program at different times:**

  When we use colour labels, we have the same probability of predicting PC_jcorrectly when it inherits the predictor state from PC_i as described in equation A.3:

  $$P(PC_j \text{ predicted correctly}) = 2p_Ap_B - p_A - p_B + 1$$

  However, since this is maximised when $p_A = p_B$, our labelling scheme increases the chance of this happening. While the distribution spans reals, the lower variance in both classes means we reduce the expected value of $|p_A - p_B|$ and so increase the expectation of $P(PC_j$ predicted correctly). Overall, we have shown that using colour labels to split branches into takenness classes ***improves the prediction of branches that have inherited their predictor state***, as the local predictor has less warming up to do.

Hence, we have shown in all 2 cases, ***takenness colour labels provide a statistical advantage for improving the predictor accuracy***.

List of Figures

Bibliography

[1] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. *SIGOPS Oper. Syst. Rev.*, 30(5):128–137, sep 1996.

[2] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, sep 2006.

[3] Andrei-Florin Sburlan. Discovering predictive patterns: A study of contextual factors for next generation branch predictors. 2023.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[5] Bobby R. Bruce, Ayaz Akram, Hoa Nguyen, Kyle Roarty, Mahyar Samani, Marjan Friborz, Trivikram Reddy, Matthew D. Sinclair, and Jason Lowe-Power. Enabling reproducible and agile full-system simulation. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, 2021.

[6] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N. Udipi. Simulating dram controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–210, 2014.

[7] Branch predictor. *Wikipedia*, Oct 2023.

[8] Kaveh Aasaraai. Computational and storage based power and performance optimizations for highly accurate branch predictors relying on neural networks. 2007.

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

[10] Andre Seznec. Tage-sc-l branch predictors. *JILP-Championship Branch Prediction*, 2014.

[11] Scott McFarling. Combining branch predictors. *Digital Western Research Lab (WRL) Technical Report, TN-36*, Jun 1993.

[12] Lin CK and Tara SJ. Branch prediction is not a solved problem: Measurements, opportunities, and future directions. *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019.

[13] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, 1997.

[14] Rinu Joseph. A survey of deep learning techniques for dynamic branch prediction. 12 2021.

[15] D.A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. pages 197–206, 2001.

[16] Shaofei Ren, Guorong Chen, Tiange Li, Qijun Chen, and Shaofan Li. A deep learning-based computational algorithm for identifying damage load condition: An artificial intelligence inverse problem solution for failure analysis. *Computer Modeling in Engineering Sciences*, 117:287–307, 12 2018.

[17] H. Akkary, S.T. Srinivasan, R. Koltur, Y. Patil, and W. Refaai. Perceptron-based branch confidence estimation. pages 265–265, 2004.

[18] Ian Cutress. Amd gives more zen details: Ryzen, 3.4 ghz+, nvme, neural net prediction, 25 mhz boost steps, 2016.

[19] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. pages 251–262, 2000.

[20] D. Grunwald, D. Lindsay, and B. Zorn. Static methods in hybrid branch prediction. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, pages 222–229, 1998.

[21] Hans Vandierendonck, V. Desmet, and Koen De Bosschere. *Behavior-Based Branch Prediction by Dynamically Clustering Branch Instructions*, volume 24. 01 2006.

[22] Tse-Yu Yeh and Harshvardhan P Sharangpani. Method and apparatus for branch prediction using first and second level branch prediction tables, 1998.

[23] Intel. Ia-32 intel® architecture software developer's manual volume 2: Instruction set reference, 2002.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023.

[25] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, page 318–319, New York, NY, USA, 2003. Association for Computing Machinery.

[26] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGPLAN Not.*, 37(10):45–57, oct 2002.

[27] *Dirichlet Box Principle*. Encyclopedia of Mathematics, 2020.

[28] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. Finding community structure in very large networks. *Phys. Rev. E*, 70:066111, Dec 2004.

[29] Ali Hassani, Steven Walton, Jiachen Li, Shen Li, and Humphrey Shi. Neighborhood attention transformer, 2023.

[30] Gang Li, Di Xu, Xing Cheng, Lingyu Si, and Changwen Zheng. Simvit: Exploring a simple vision transformer with sliding windows, 2021.

[31] Ildar Abdrakhmanov, Evgenii Kanin, Sergei Boronin, Evgeny Burnaev, and Andrei Osiptsov. Development of deep transformer-based models for long-term prediction of transient production of oil wells. 10 2021.

[32] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

[33] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

[34] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. Quantifying the carbon emissions of machine learning. *arXiv preprint arXiv:1910.09700*, 2019.

[35] *Law of Large Numbers*, pages 297–299. Springer New York, New York, NY, 2008.