Imperial College
London

MEng Individual Project

Imperial College London

Department of Computing

Discovering Predictive Patterns: A Study of Contextual Factors for Next Generation Branch Predictors

Author:
Andrei-Florin Sburlan

Supervisor:
Prof. Paul Kelly

Second Marker:
Dr Giuliano Casale

June 19, 2023

Abstract

Branch prediction is a fundamental part of every modern high-performance processor. However, in spite of its great importance, breakthroughs in this field have been rare. For a long time now state-of-the art predictors have based their decisions solely the recent taken/not taken branch history, disregarding other types of context.

This project is centred around exploring new ways of improving branch prediction, in particular by augmenting previous methods with extra information about the program execution. We begin by analysing the effectiveness of a number of context types derived from the Return Address Stack, though we also dive into the field of Machine Learning by introducing the novel perspective of branch prediction as a Machine Translation problem. In light of this, we train a Transformer Neural Network predictor, show its effectiveness in a comparative study with other approaches and finish by illustrating how analysing the internal weights of the model can give insight into subtle branch correlations. We hence arrive at a number of key findings:

- Integrating the Return Address Stack into branch prediction has the potential to reduce the misprediction rate by up to 12% of its initial value
- For object oriented languages, the Return Address Stack at the time of object allocation is predictive for branch outcomes in virtual methods
- Transformer Neural Networks can be successfully trained to predict branch outcomes. Moreover, their internal weights can be examined to gain insight into subtle branch correlations.

Acknowledgements

I would like to begin by thanking my project supervisor, Prof. Paul Kelly, for his guidance and for always encouraging me to pursue the research directions that I found most exciting.

I would also like to thank my family for teaching me the values that made me who I am today.

Last but not least, I would like to thank my friends, who have always supported me in my pursuits.

Contents

List of Acronyms

Chapter 1

Introduction

Ever since the advent of computing, there has always been a desire for faster execution of programs. For many practical workloads though, this often requires accelerating just a single thread of execution. Under this light, one of the main goals of processor architects is to maximise the instruction execution throughput of their designs.

A cornerstone idea in this area is processor pipelining, where execution is split in many, independent stages such as fetching, decoding, arithmetic execution or storing results. Instructions progress through the pipeline one stage at a time: as one instruction completes a stage, the next can take its place. Thus multiple instructions are allowed to be in-flight within the processor at the same time and maximum throughput is achieved when each stage is occupied. Should a stage not be able to make progress, a pipeline bubble will occur, which results in decreased performance as subsequent stages will also experience a stall in the future.

Branch Prediction is a technique commonly used in modern processors that aims to anticipate the outcome of branches within a program before they have actually finished executing. Its purpose is to alleviate potential stalls which would otherwise occur if the processor waited for branches to be fully executed before fetching subsequent instructions. However, the accuracy of the Branch Prediction Unit (BPU) greatly influences the performance of such designs, as any miss-speculation results in fetching erroneous instructions whose execution will eventually be thrown away.

The aim of this thesis is to evaluate how extra information could help improve the performance of branch predictors. Most state-of-the-art BPU designs largely base their decisions on recent branch outcomes. Our initial hypothesis is that the behaviour of branch instructions inside a program is highly dependent on the current context of the program as well as the micro-architectural state of the processor. Although these provide for a very rich stream of information, identifying the relevant state that correlates with certain branch outcomes is far from trivial. In this work, we will present our methodology for extracting meaning out of this extra context, as well as evaluate its predictive value for use in future branch predictors.

1.1 Contributions

In the following chapters, we will present a number of approaches that aim to improve branch prediction by means of using more information about the program execution. In order, these are:

- **Return Address Stack Correlations**: we provide an extensive analysis into the predictive power of the micro-architectural Return Address Stack (RAS) for anticipating branch outcomes by examining the statistics captured in a set of traces that we collected by running a number of benchmark programs on a micro-architectural processor simulator.
- **Object Allocation Context Correlations**: similarly to the last point, we examine whether programs built from object oriented languages exhibit branches that are correlated with the return address stack at the time of object creation. This is a more complex analyses, as it requires modifying both the compiler and the simulator.
- **Transformer-based Neural Branch Prediction**: Ever since the introduction of the Transformer neural network architecture, we have seen a true paradigm shift in the field of Natural Language Processing (NLP). This architecture is the current state-of-the-art when it comes to many tasks in NLP (particularly those related to text generation), and a large part of its success is due to its unique ability to automatically identify correlations between the tokens of a sequence. Inspired by the recent developments in NLP, we introduce a novel take on branch prediction, inspired by machine translation, and show that Transformers can be successfully used to predict branch outcomes with great accuracy, although with potentially unrealistic hardware requirements.

Chapter 2

Preliminaries

In this chapter, we will outline some key computer architecture concepts which will set the background for the problem of branch prediction. We begin by reviewing the structure of common branch instructions in the ARM AArch64 Instruction Set Architecture (ISA) followed by an analysis of their interaction with different Central Processing Unit (CPU) architectures. Finally, we provide an overview of the fundamentals of branch prediction, introducing key terminology that will be used throughout the report.

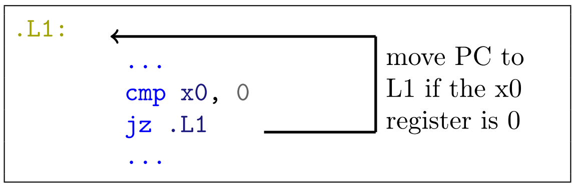
2.1 Branch Instructions

Control flow is an essential part of any program. Be it functional, procedural, object-oriented or anything in between, any programming language supports some notion of conditional statements and loops. These structures are built over branching instructions, reflected in every ISA. In the following two subsections, we will briefly survey the most common branch instruction types in ARM's AArch64 instruction set, which we will be using in this report.

2.1.1 Conditional Branch Instructions

Conditional Branches have the effect of moving the Program Counter (PC) to an address specified in the instruction, subject to a condition being true (e.g. a status register is zero). They are often used to implement control flow structures such as *if statements* or the exit conditions for *loops*.

Execution proceeds as normal if the condition is false: the program counter is simply moved to the next adjacent instruction in memory. BPUs have the job of predicting their *takendness*, even if the condition is still being evaluated.

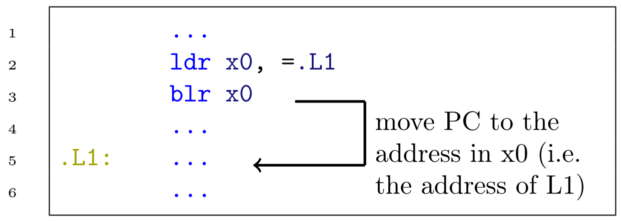


```
.L1:
        ...
        cmp x0, 0
        jz .L1
        ...
```

Figure 2.1: Example of Conditional Branch Instruction in AArch64

2.1.2 Indirect Branch Instructions

Indirect Branches have the effect of moving the program counter to an address specified in a register. One of their most common applications is in the implementation of function pointers. These are commonly used in the context of Object Oriented Programming when a program has to select the next method to execute based on an object's type i.e. a virtual call. BPUs have the job of predicting their address, even if the value in the address register has not yet been computed.



```
        ...
        ldr x0, =.L1
        blr x0
        ...
.L1:    ...
        ...
```

Figure 2.2: Example of Indirect Branch Instruction in AArch64

2.2 Processor Types

Most processors today can be broadly classified in two types: in-order and out-of-order, with the former being more common in low power applications while the latter offering much higher performance at the cost of energy usage. Nevertheless, they all employ variants of branch predictors.

2.2.1 In-Order Processors

In-order processors are built as a linear array of stages. All instructions progress through all stages one at a time and in their natural order. Often times, their first stage is tasked only with fetching the next instruction from memory. One can easily see here how branches can be problematic in this context: it may well be the case that the branch depends on another instruction that is still in the pipeline, hence its result is not yet known. Branch prediction is used in this case so as to remove an unnecessary pipeline stall. Should the prediction be deemed wrong at a later time though, the pipeline has to be flushed and the processor has to be reverted to its last valid state - an expensive operation which can greatly reduce performance.

2.2.2 Out-of-Order Processors

Out-of-order processors take the idea even further, by dynamically reordering instructions in their pipeline to make better use of the machine's resources. Their pipelines are generally much more involved and contain many paths of execution to allow a greater degree of parallelism between instructions. They have much higher throughput and, in turn, also suffer much more from wrong branch predictions, as the processor might have made much more progress on an incorrect execution path and the rollback will be more expensive.

2.3 Basic Branch Prediction

In this section, we will give an overview of the fundamentals of branch prediction.

2.3.1 Static Branch Prediction

Static methods are perhaps the simplest ways of performing branch prediction. Instead of recording branch outcomes during run time and making predictions according to the collected data (i.e. *dynamic prediction*), static predictors have fixed outcomes for all branches. Examples include *the always taken predictor* (conditional branches are always taken), *the always not taken predictor* (conditional branches are always not taken), or predictors that include static branch hints (e.g. the ISA implements branch instructions that include information about how they should be predicted).

2.3.2 Bimodal Predictor

The Bimodal Predictor is a building block of many more advanced BPUs. It is implemented as a saturating counter, whose most significan bit denotes its current prediction. This behaviour is illustrated as a state machine in Figure 2.3.

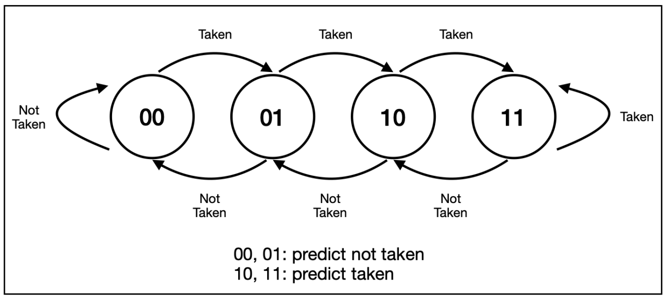


Figure 2.3: 2-bit Bimodal Predictor as a State Machine

2.3.3 Branch History Tables

The Branch History Table (BHT) is implemented as an array of Bimodal Predictors. The array is normally indexed by a number of lower bits of the Program Counter as depicted in Figure 2.4. Although often used as a building block for other predictors, the BHT can be used by itself to implement a simple local branch predictor, i.e. a predictor that only uses local information about each branch when making decisions.

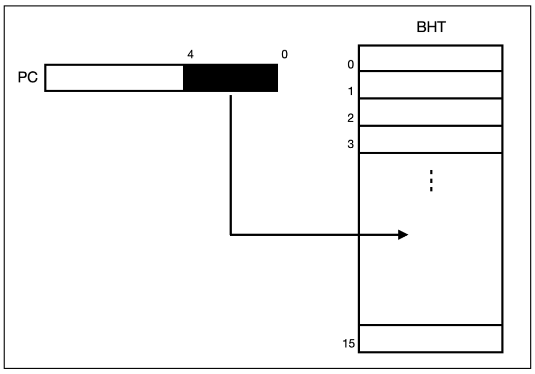


Figure 2.4: Branch History Table Structure using 4 PC bits

2.3.4 Taken History and Two Level Predictors

The Branch History Register (BHR) is commonly used when indexing BPU structures. It is implemented as a shift register and it is updated with every branch decision, forming the taken history. The GSelect predictor best illustrates how it can be used in practice: GSelect employs an array of Branch History Tables. Based on the BHR, it selects an appropriate BHT which is then indexed via the PC. This sort of scheme is also known as a *two level predictor*, as indexing is done in two steps: once through the BHR and then through the PC.

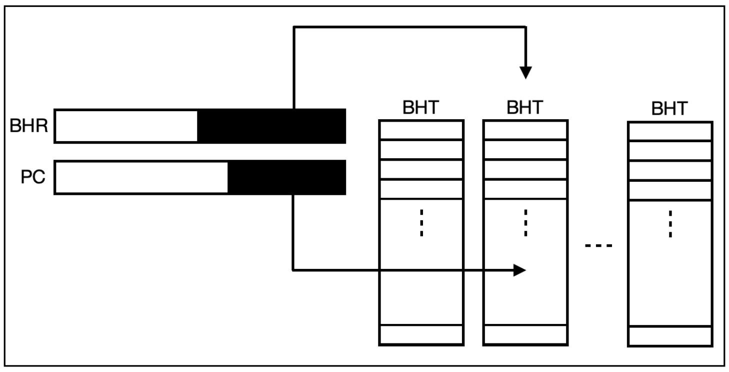


Figure 2.5: GSelect Predictor

2.4 The Predictive Power of the Taken History

As used in the GSelect predictor, the taken history may appear at first as a rather arbitrary choice of context to base branch prediction upon. However, the idea has proven highly successful, with many of the best predictors created in recent times being based upon it. In this section, we aim to provide some intuition for why the taken history is used in branch prediction.

```
if (condition A) {
    ...
}
...
if (condition A && condition B) {
    ...
}

```

(a) Correlation Through Branch Conditions

```
if (...) {
    ...
    a = False;
}
...
if (a && condition B) {
    ...
}
```

(b) Correlation Through Branch Bodies

Figure 2.6: Simple Example of Correlated Branches

Consider, for the sake of argument the two code listings in Figure 2.6. Clearly, the two examples showcase correlated branches. In the former, the branches are correlated by virtue of sharing a common term in their conditions (`"condition A"`). Assuming the code in between the branches has no effect on the value of `condition A`, then if the first branch is not taken, the second one will not be either. In the latter example, the branches are correlated through the effect of the body of the first `if` statement: it sets the variable `a` to `False`, resulting in the second branch not being taken.

Although these examples are rather simplistic and easy to identify statically, they do showcase how branches could be correlated in practice. Moreover, with more complex programs, such correlations become very subtle and hence hard to recognise. It is under this light that the beauty of correlating branch outcomes with the recent branch history becomes apparent. The shared characteristic between the two examples above is that in both cases, the execution of the inner body of the `if` statement is predictive for the second `if` - in compiler terminology, such continuous blocks of instructions that do not contain branches are known as *Basic Blocks*. Therefore, when predictors such as GSelect examine the recent taken history, they in fact approximate which basic blocks have been executed recently in the program. Hence, without ever needing to analyse the semantics of the code, they compute statistics for branch outcomes based on this context, correctly capturing many cases such as the ones showed above.

Chapter 3

Background

The aim of this chapter is to conduct an extensive review of the academic literature on the topic of branch prediction, with particular emphasis on research pertaining to using more contextual information when making predictions. In order, we will cover the current state-of-the-art in terms of Branch Prediction Units, followed by a discussion of the future of this field and its limits. We end by outlining other research done in the area of usign the path history for prediction.

3.1 Current State-of-the-Art Predictors

In the following four sections we will survey the state of the art in terms of publicly available branch predictor designs. Our analysis and the types of predictors covered are inspired by a similar overview done by researchers at Intel [1].

3.1.1 TAGE Predictor

The TAGE predictor [2] is widely regarded as one of the most performant branch predictors. Being itself a descendent of other well established predictors (O-GEHL and PPM), it joins together some of the most successful and influential ideas in the field.

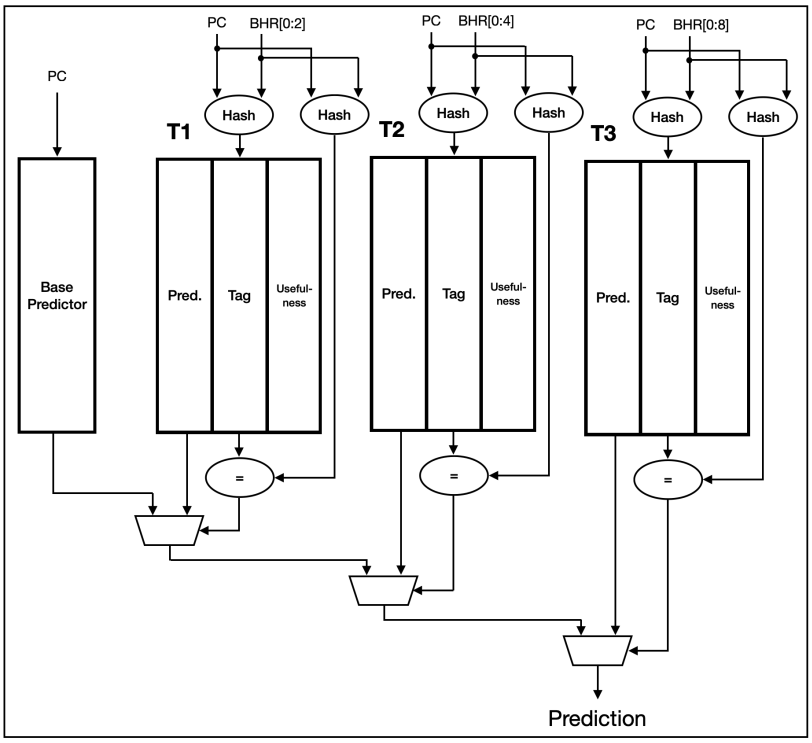


Figure 3.1: The Structure of the TAGE Predictor (based on a diagram from [2])

As showcased in Figure 3.1, TAGE is implemented by simultaneously indexing a base predictor **T0** alongside a number of custom predictor tables **Ti**. **T0** is implemented as a standard Branch History Table. On the other hand, the **Ti** predictors use a more complicated scheme. They are implemented as tables of Bimodal Predictors, indexed and tagged through a hash of the PC and the BHR. Crucially, indexing the **Ti**'s uses history lengths following an arbitrary geometric series. As such, TAGE is able to use long histories for indexing while also making good utilisation of its storage capacity. The final prediction is given by the match corresponding to the longest history or the base predictor if there are no matches. Finally, the **Ti** predictor entries also maintain an *usefulness* counter used for entry updating, allocation and deallocation.

TAGE and its variants proved their performance by winning several branch prediction contests, amongst them being the second and third editions of *The Championship Branch Prediction* contest. It has also seen use in practical designs e.g. recently within the BOOMv3 Core [3].

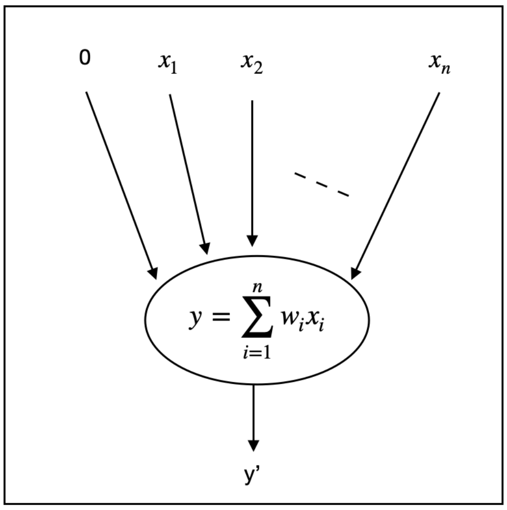
3.1.2 Machine Learning Predictors

Machine Learning (ML) is a field concerned with creating algorithms that *learn* to perform certain tasks by observing data rather than being explicitly programmed to solve them. In recent times tremendous progress has been made in this field, ML methods providing great solutions to many problems previously deemed untractable.

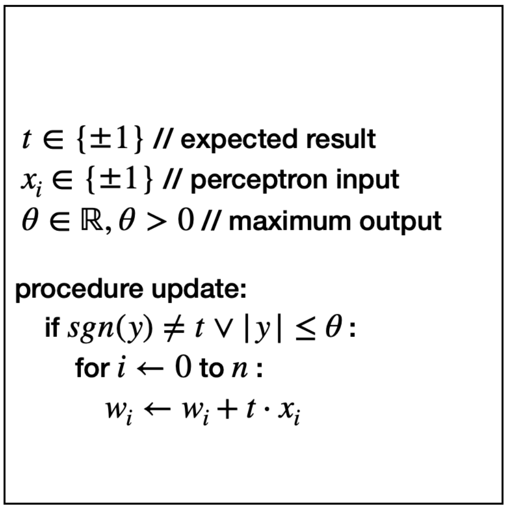
One of the driving forces of this revolution has been the research into Artificial Neural Networks, which are function approximators inspired by the inner workings of biological neural networks. It has been shown in the past that neural networks are universal approximators [4] - i.e. given enough capacity, they are able to approximate any continuous function to arbitrary precision. This generality paired with efficient numerical training techniques proved to be a strong baseline for many learning algorithms. However, although many problems can be solved by neural networks, it should be noted that this often necessitates the usage of large models, which can be expensive and time consuming to train.

Given the complexity of branch behaviour, a natural research question would be whether machine learning techniques (and specifically neural networks) could be used for this problem as well. Computer architecture comes with a new set of challenges though: any branch prediction algorithm would eventually need to be implemented in hardware. As such, large models are challenging to deploy in practice, partially due to their expensive silicon implementation but also due to their slow learning, which would make them impractical for small workloads.

Even so, research has explored other avenues of machine learning, and Perceptron Networks (a precursor to regular neural networks) have proven to be realistic candidates for prediction units, as shown by Jiménezg and Lin [5].



(a) Perceptron Prediction



```
$t \in \{\pm 1\}$ // expected result
$x_i \in \{\pm 1\}$ // perceptron input
$\theta \in \mathbb{R}, \theta > 0$ // maximum output

procedure update:
  if $sgn(y) \neq t \lor |y| \leq \theta$:
    for $i \leftarrow 0$ to $n$:
      $w_i \leftarrow w_i + t \cdot x_i$
```

(b) Perceptron Update Procedure

Figure 3.2: The Inner Workings of a Perceptron (based on [5])

Figure 3.2a shows a graphical representation of a perceptron. A prediction is made by computing a weighted sum of inputs: if the result is positive, the perceptron outputs a 1, otherwise a -1. A perceptron is trained by learning the weights used in the aforemetioned sum. Initially, the weights take random values, but incremental updates can be applied from *(input, output)* samples from the target function: Figure 3.2b.

The approach described by Jiménezg and Lin builds an array similar to a Branch History Table that replaces the saturating counters with perceptrons. Perceptron inference is done taking recent branch history as input and the output is interpreted as 1 corresponding to *predict taken* and -1 to *predict not taken*. Upon learning the true outcome of a branch, the predictor performs one update step.

Finally, the authors evaluate their new predictor on the SPEC 2000 benchmark suite. Although benchmark suites can provide a general expectation for performance, the field of architecture has moved significantly since the paper's publication in 2001. Still, this general approach has found significant success in industry: AMD, one of the largest designers of x86 CPUs has stated that the branch predictor of their Ryzen architecture uses *Neural Net Prediction* [6], suspected to be in fact a perceptron-based approach [7].

3.1.3 Domain Specific Predictors

Although general-purpose branch predictors are in principle capable of handling any branch, realistic programs often exhibit certain execution patterns (e.g. simple *for-loops* with fixed bounds). Further optimising such cases can prove highly beneficial, especially given that custom hardware can be design specifically for their use case. Consequently, the research literature has also explored Domain Specific Predictors tailored for simple control flow patterns (e.g. loop predictors).

3.1.4 Tournament Predictors

Tournament Predictors aim to combine multiple predictors into a single monolithic one such that their strengths would cover the weaknesses of each individual approach. Often times extra hardware has to be introduced to select the appropriate branch predictor to use, as is the case in TAGE-SC-L [8]: a pairing of TAGE with a loop predictor and a Statistical Correlator to choose between the two.

3.2 Branch Prediction - a problem not yet solved

Branch Predictors have been studied for a long time. As such, there have been many iterations over these concepts making current designs surprisingly proficient at learning and guessing branch outcomes. State of the art predictors such as TAGE-SC-L have been evaluated on the *Championship Branch Prediction* benchmark programs and achieved less than 3.3 missed predictions per 1000 instructions (3.3 MPKI) [8] - certainly an impressive result which could hint that this topic has been largely solved.

Alas, in practice their performance varies. Indeed, the patterns found in many common applications are captured by current predictors, yet one can definitely find real workloads for which their performance leaves room for improvement. This observation is best showcased in a recent 2019 study by Chit-Kwan Lin and Stephen J. Tarsa at Intel Corporation, suggestively titled "Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions" [1]. In this paper, the authors show why improvements to branch prediction could have a very significant impact on CPU performance, potentially on par with advancing to a new silicon technology process.

Their analysis is based on a simulated processor with an Intel Skylake configuration (an out-of-order core) and a TAGE-SC-L predictor, whose pipeline is progressively widened (i.e. it allows for more instruction level parallelism). As a first step, they classify poorly predicted branches in two categories:

- Hard-to-predict branches (H2Ps): branches which are inherently hard to model with current methods, yet are commonly encountered.

- Rare branches: branches which are very rarely executed; it is somewhat expected for these to have poor performance as BPUs are trained *after* encountering branch instructions.

Upon running the SPECint2017 benchmark suite, they find that performance quickly plateaus, regardless of the branch predictor capacity. In stark contrast to this, running the simulation with "perfect" or "perfect H2P" branch prediction shows great potential Instructions per Cycle (IPC) increases - e.g. a 4x pipeline scaling yields a 55.3% increase in IPC. This evidence strongly suggests that any improvement in H2P prediction could make wider pipelines more feasible and thus yield great performance gains.

3.3 Theoretical Limits of Branch Prediction

Another important angle from which the problem of branch prediction should be approached is its theoretical limit: given that BPUs are trained with limited information and only on past branch outcomes, some amount of miss-predictions are inevitable to occur and hence a *perfect* branch predictor cannot exits.

Chen, Coffey and Mudge have previously explored this problem through the lens of data compression theory [9]. Their paper starts by introducing a conceptual model for a BPU, split in three main parts as follows:

- **The source**: the machine code which is actually being executed.
- **The Information Processor**: this unit has the job of observing *the source* as it is being executed. Any relevant information (e.g. branch history) is then encoded and dispatched to a selected *predictor*.
- **The Predictor**: a simple state machine that produces an output prediction given the input from the information processor. Note that the predictor does not need to understand the meaning of its input.

Most BPUs fall within this model. For example, in the case of GSelect, *the information processor* uses recent recent branch history and the program counter to select a *predictor* implemented as a saturating counter. The counter then observes the branch outcome once it is known.

The main contribution of this paper is the introduction of the Prediction by Partial Matching (PPM) predictor, which is derived from the compression algorithm with the same name. In broad terms, lossless data compression tries to assign bit strings to letters within a text based on their frequency. Common letters are assigned shorter encodings, which in turn reduces the length of the compressed text. An essential part of most data compression algorithms is the derivation of the data's statistics i.e. given a context text, one has to predict the following letter. As such, there is a striking similarity between this problem and branch prediction: most state-of-the-art branch predictors try to predict the next branch outcome given the Global History Register. Moreover, the PPM data compression algorithm has been shown to be optimal, therefore using it in the context of branch prediction should provide an upper bound for BPU performance.

The authors show that the PPM predictor does fall within their branch predictor model (though it may be unrealistic to implement in practice due to its resource requirements). For brevity, we will omit its inner workings, as for our purposes it only matters that optimal performance is achieved. Even so, the authors show that the improvement is only marginal: current predictors asymptotically approach the optimal performance of PPM given enough capacity and time.

The implications of this study are therefore very profound: *great leaps in branch predictor performance are now only possible by consuming more information than previous designs* (i.e. beyond recent branch history).

3.4 Path History

3.4.1 Loop Code Replication

Andreas Krall has explored the the concept of *code replication* in his paper on improving semi-static branch prediction [10]. Thus far, we have explored *Dynamic Branch Prediction*, where a processor contains special hardware that trains itself to predict branch outcomes during run time. In this paper, Krall chooses to focus on *semi-static branch prediction*, where program profiling is used to make decisions about branch takendness before run time (in this context, profiling data is used to select a single static prediction for each branch). The paper examines in detail ways in which intra loop branches (branches inside a loop) could be better predicted. Semi-static approaches are hard to apply in such circumstances because the behaviour of intra loop branches could be correlated with previous iterations.

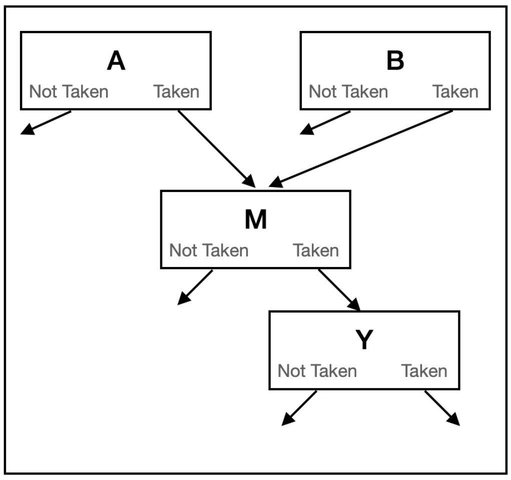
Taking inspiration from dynamic methods, the author chooses to focus on a single branch within a loop, for which profiling data is used to create a history pattern table (i.e. a table which associates different histories to branch outcomes). It should be noted that such tables can potentially become unrealistically large (there are 2^n patterns associated with histories of length n), though the author addresses this issue by exemplifying common programs which only exhibit a small subset of all possible histories.

With this information, the paper then details a way of replicating the loop body to allow semi-static branch prediction. Each replication corresponds to a different state within a finite automaton. The automaton's transitions are based on branch decisions such that the states model the branch history. As such, each copy of the initial branch should have an associated static prediction based on the profiled data. Finally, the target addresses of each branch copy correspond to transitions in the finite automaton.

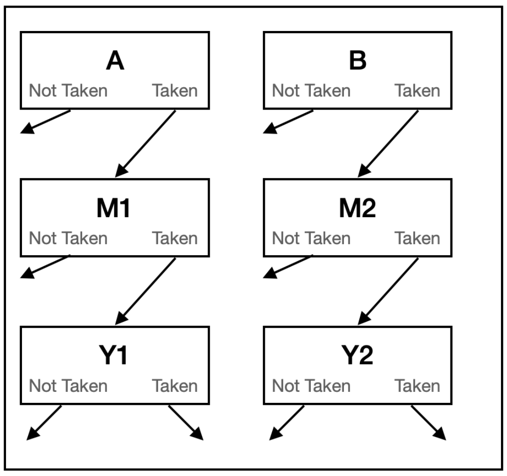
In effect, the author hence introduces a scheme which allows static prediction at the cost of increased code size. Naturally, this approach is only beneficial provided that the profiled data is relevant for many executions of the program, though the paper's analysis suggests that this is indeed the case and performance can sometimes surpass that of dynamic methods.

3.4.2 Path Cloning

In a related study, Young and Smith show a different approach the same problem of semi-static branch prediction [11]. Their paper details a novel profile guided compiler optimisation that aims to improve static predictions by means of cloning parts of a program's control flow graph. To illustrate their idea, consider Figure 3.3.



(a) Before Cloning



(b) After Cloning

Figure 3.3: Example illustrating the benefit of path history over pattern history (based on diagrams in [11])

Figure 3.3a shows a number of nodes within the control flow graph of a program. In this simple example, in order to reach the node Y, execution has to follow one of two paths: AMY or BMY. An essential aspect to consider is that both paths would appear as having the same branch history. Moreover, it may be the case that the branch outcome at node Y is correlated with the path taken, e.g. on path AMY the branch at Y is likely taken whereas on path on path BMY it is likely not taken. Such patterns may hinder the accuracy of a history based dynamic branch predictor. The solution detailed in the paper is to use profile data to decide when to clone the control flow graph, as shown in Figure 3.3b. Such an optimisation could be of help for both dynamic and static predictors, though it should be noted that using it aggressively could lead to large increases in binary size which in turn could decrease performance.

Chapter 4

Execution Trace Generation

In this chapter, we present our methodology for analysing the characteristics of several representative benchmark programs by running them in simulation and capturing the relevant micro-architectural state. The resulting execution traces will serve as the foundation for our analysis in the subsequent parts of this work.

4.1 Trace Generation

4.1.1 Simulation Environment

For the purposes of this project, we have chosen to use the Gem5 processor simulator because it strikes a good balance between micro-architectural detail and simulation speed. Gem5 provides a state-of-the-art processor model with an out-of-order execution pipeline. Furthermore, it has support for many branch predictor topologies - of particular interest for our purposes is the aforementioned TAGE predictor due to its wide commercial deployment.

Specifically, we have configured Gem5 (version 22.1.0.0) with the following parameters:

- Simulation environment uses system call emulation: we choose not to model the execution of an operating system, opting to forward system calls to the underlying host of Gem5. We believe this is a sensible choice as for our purposes we are most interested in the performance characteristics of a single program with a single thread of execution.

- ARM processor frontend: Gem5's processor model is largely ISA independent, with the exception of its frontend that decodes instructions into micro-ops. As such, we use the appropriate frontend for our chosen ISA.

- Cache hierarchy with 64KB L1 and L2 caches.

- TAGE-SC-L branch predictor with 64KB capacity.

- Other Gem5 parameters remain to their default values, as they are already configured to resemble an Alpha 21264 processor.

4.1.2 Benchmark Programs

As covered earlier, the aim of this project is to evaluate possible opportunities to improve branch prediction in real workloads. As such, we opted to analyse the SPEC CPU 2006 benchmark suite as it provides a varied suite of programs used in practice. Moreover, we will limit our analysis only to the *Integer* benchmarks, hence omitting the *Floating Point* ones, as the former are more focused on intricate control flow whereas the latter tend to be centred around applications with high arithmetic intensity. To compile the programs, we used the Clang 16.0.2 compiler suite, with O3 optimisation level as well as link time optimisation enabled, as these choices should be representative of how production binaries are built in practice.

4.1.3 Execution Trace Generation

The execution trace was extracted by instrumenting Gem5's processor pipeline in key stages such that relevant micro-architectural state may be captured. This was achieved by augmenting the model with a new abstract module with an ISA-agnostic interface, the *BranchTracer*, which is called at two key execution phases: firstly, it tags every branching instruction passing through the fetch stage with a record containing a snapshot of the processor state (i.e. registers, return address stack) just before using the BPU as well as its initial prediction and secondly, in the commit stage, it logs the previously captured state for every instruction that eventually exits the pipeline (at this later stage we also capture the eventual outcome of the branch). As such, instructions executed due to miss-speculations are never logged, which is the desired behaviour. Figure 4.1 shows a simplified view of Gem5's processor pipeline, as well as the additions made to the Fetch and Commit stages.

However, it should be noted that in practice it is quite common for real programs to run in excess of several millions of instructions, making these captured traces challenging to process and store. As a consequence to the way the data is generated (i.e. sequentially as instructions get simulated), the natural way to save it is row-wise i.e. saving each data point as a row to some table. Furthermore, Gem5's logging framework does not provide infrastructure for using more advanced storage techniques, so we initially opted to save the traces as simple *Comma Separated Values* (CSV) files, although this proved to be rather inefficient for a number of reasons. Firstly, CSV files do not record any type information of the data, which makes them particularly unsuitable for storing large tables as they make poor utilisation of the disk space: indeed we have found that even small programs yielded traces of several GBs. More importantly though, we are most interested in running large aggregations to gain insight into the statistics of this large volume of data that we are generating. More often than not, this only requires iterating and filtering on some subset of the columns in our data, which is unfortunately impractical for large CSVs due to high RAM and CPU time usage. To address these issues, we implemented a new translation layer on top of the Gem5 infrastructure to convert the CSV data in real time to the more Apache Parquet format - a typed column-major format that employs powerful data compression (in our case, up to 10x trace size reduction). In Table 4.1 we present the format of the resulting execution trace, where each entry represents a committed branch instruction.

Name	Type	Description
tick	uint64	The simulation tick at which the instruction was reached.
disassembly	string	Disassembly of the instruction.
inst_addr	uint64	The virtual address of the branch instruction.
inst_rel_addr	string	The address of the instruction as an offset to the closest function symbol (e.g. "foo+12").
pred_addr	uint64	The virtual address to which the branch was predicted to jump to.
pred_rel_addr	uint64	The address to which the branch was predicted to jump to, as an offset to the closest function symbol.
jump_addr	uint64	The address to which the branch actually jumped to.
jump_rel_addr	string	The address to which the branch actually jumped to, as an offset to the closest function symbol.
pred_taken	bool	Whether the branch was predicted as taken.
taken	bool	Whether the branch was actually taken.
virtual	bool	Whether the branch represents an indirect jump.
return	bool	Whether the branch is a return instruction.
call	bool	Whether the branch is a call instruction.
ras	List<uint64>	The micro-architectural return address stack.
ras_rel	List<string>	The micro-architectural return address stack, with addresses as offsets to the closest function symbol.

Table 4.1: Trace Fields Captured from Gem5 Simulations

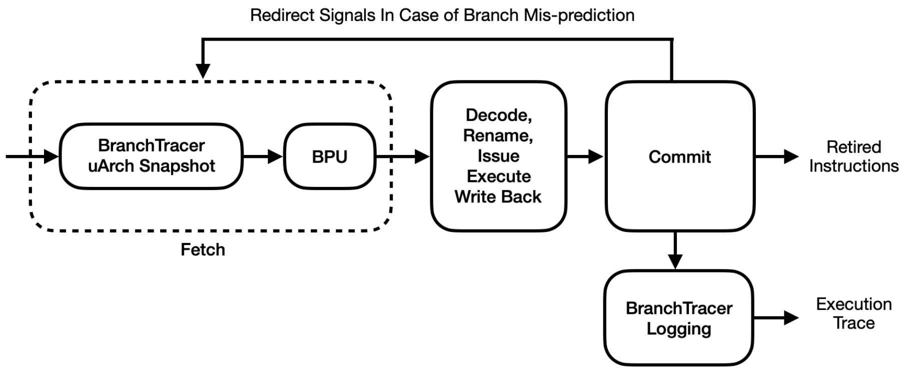

Redirect Signals In Case of Branch Mis-prediction
BranchTracer uArch Snapshot
BPU
Fetch
Decode, Rename, Issue Execute Write Back
Commit
Retired Instructions
BranchTracer Logging
Execution Trace

Figure 4.1: Simplified view of Gem5's Processor Pipeline After Adding the BranchTracer

4.2 Generating SimPoints

A big challenge when researching computer architecture by means of simulation is that real world programs that exhibit interesting behaviour often tend to run for several billions of cycles. Naturally, this translates to very long simulation times. In our case, we have found benchmarks that take up to a couple of days to simulate - the specifics of our hardware are not relevant for this argument, but rather it is important to stress that these long simulation times would have greatly hindered our ability to iterate quickly on different ideas.

4.2.1 Overview

SimPoints [12] provide for an elegant alternative to simulating programs in full. To begin with, the paper's authors start by observing that at run time programs often exhibit recurrent behaviours as they go through their different stages of execution. As such, SimPoint tries to identify a small but representative set of samples out of the full program execution that would faithfully represent it.

More formally, in SimPoint we begin by splitting the execution trace of a program in contiguous intervals of a set size (e.g. one million instructions). For each interval, a feature vector is built based on which basic blocks are executed within it and upon clustering them, the phases of a program are discovered. Following this procedure, an interval is sampled from each cluster and a weight is assigned to it based on the cluster size. The selected intervals can then be run in simulation, and upon weighting the resulting statistics, we should attain great approximations to the full program execution statistics in a fraction of the time.

4.2.2 Warm Up and Interval Length

An important thing to note when building simpoints is that ultimately they only represent an approximation of the whole program execution (though many times a surprisingly good one). Yet, there are a number of hyper-parameters to the algorithm that can greatly impact its performance. As reported in the paper, there is a subtle balance to be maintained between the interval length, the maximum number of clusters allowed and simulation time for the simpoints. There are advantages and disadvantages to both long and short interval lengths, as they provide different levels of granularity that can be desirable for different applications. Furthermore, they are very closely related to the maximum number of clusters allowed, as normally the optimal number of clusters often increases with shorter intervals. There is also the issue of warm up: since only selected intervals are simulated, short traces can have their statistics skewed by cold micro-architectural structures such as caches.

4.2.3 Tracing and Reporting Methodology

In this work, we use Gem5's built in support for generating the feature vectors and the *SimPoint* tool (as provided in the paper) to perform the clustering, with the values recommended by the paper of 10 million instructions per interval with a maximum of 30 clusters, over which we also add a warmup period of 1 million instructions. Furthermore, we use Gem5's simplest (and fastest to execute) processor model (NonCachingSimpleCPU) to build simulation checkpoints for the intervals representative of the program's phases. The checkpoints are then resumed using the highly detailed out of order core, which then generates our custom execution traces. Finally, when reporting our results regarding the execution of some program, we present a weighted average of the statistics of each SimPoint interval.

Chapter 5

Simple Micro-Architectural Correlations

In this chapter, we evaluate the predictive power of simple micro-architectural contexts with regards to branch outcomes, with the aim of integrating this knowledge either in future hardware predictors or in software optimisations within the compiler.

We hence take as a starting point the hypothesis that the branch outcomes within a function are correlated with some easily accessible micro-architectural context (e.g. the call stack). Under this light, functions that are reached through many execution paths are of particular interest, as they could become candidates for further optimisations such as cloning (duplicating code and selectively calling it, such that physical addresses encode the context) or could simply index an extra dimension of a branch predictor table, akin to current methods.

5.1 Return Address Stack Correlation

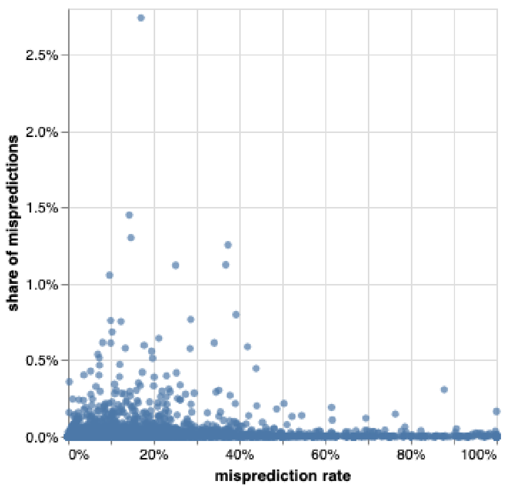
5.1.1 Overview

We begin by considering the context of the call stack. While the call stack is generally not stored directly, it can always be recovered by examining the return address of every function call. On many processor architectures (including ARM based cores), the return addresses are stored together with each stack frame, in system memory. Normally this information would be rather expensive to access in a dynamic setting, hence we turn to using a structure that is already implemented in virtually all modern processor cores: the RAS. The RAS is a simple finite-capacity stack implemented in hardware that maintains a snapshot of recent return addresses - its main purpose is to accelerate the execution of return instructions. We therefore make use of this hardware structure as it also has a correspondent within Gem5 and whenever a branch instruction is reached in simulation, we save its contents in the execution trace.

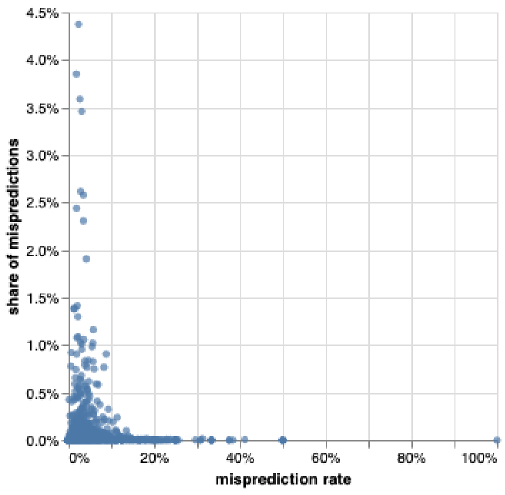
5.1.2 Case Study

In the following, we present a of case study that illustrates the effectiveness of using the RAS to enhance branch prediction. Our aim in this section is to showcase kinds of analyses that we conducted as applied to a concrete benchmark, leaving the complete results across the full SPEC2006 benchmark suite for the evaluation chapter.

We begin by examining the SPEC2006 403.GCC benchmark with the train input set. This benchmark is based on the well-known GNU Compiler Collection and consists of a standard C compilation task. Compilers are particularly interesting for analysing branch prediction as they exhibit a high density of control flow structures. As a fist step, we use SimPoints as described in Section 4.2 to approximate the global misprediction rate of TAGE 64KB on this program: 2.22% - a very low value, though as we have covered in Section 3.2, even small improvements can have profound consequences for example by allowing wider processor pipelines.



(a) Scatter Plot of All Branches in 403.GCC



(b) Scatter Plot of All Functions in 403.GCC

Figure 5.1: Global Statistics for the SPEC2006 403.GCC Benchmark

To further focus our analysis, we plot in Figure 5.1a all branches in the program, alongside their misprediction rate and total share of all mispredictions. We hence observe that most branches in the program have misprediction rates under 50%. Furthermore, we observe that the mispredictions in this program cannot be attributed only to a small set of branches: even the highest heavy hitter, only accounts for 2.74% of all mispredictions.

Similar results are achieved when grouping branches based on the function they are part of (Figure 5.1b). Although on an individual level each function contributes a relatively low fraction of all mispredictions, the program execution could benefit from optimising them. We hence considered the possibility that perhaps a simple local predictor is sufficient for predicting the branches within certain functions, provided that the return address stack is known. The main intuition behind this idea is that while functions are generally written in such a way that they handle many use scenarios, their call sites sometimes focus only on some specific cases. Furthermore, this approach has a natural correspondent as a potential compiler optimisation in the form of function cloning, where certain functions would be duplicated within a binary. Upon selectively calling these functions based on the call stack, the virtual address of the cloned branches would end up effectively encoding some information about the return address stack, thus making classical predictors such as TAGE learn different behaviours for each context.

Building on this idea of function cloning, we propose a way to calculate an upper bound for its potential for improving the misprediction rate. Firstly, we compute the taken ratio r_{taken} and misprediction rate r_{miss} for each branch / RAS context (i.e. top of the return address stack) combination in the benchmark. Based on this information, we can approximate the error of a local predictor for each branch in each context as $r_{local} = \min(r_{taken}, 1 - r_{taken})$. By filtering the trace to only the entries satisfying $r_{local} < r_{miss}$ we obtain candidate branches and contexts where cloning would be beneficial. Aggregating this data allows us to compute the change in expected misprediction rate as $\Delta r_{miss} = r_{miss} - r_{local}$ for each branch/context pair and hence the change in the global misprediction rate by appropriately weighting these rate with the total number of branch occurrences, yielding a total of change of 0.35% for this benchmark (i.e., moving the total from miss rate from 2.22% to 1.87%).

To conclude, we managed to obtain a marginal change in expected misprediction rate, though in practice there are other factors that could limit the effectiveness of this approach: replicating code in a binary will increase pressure on the internal micro-architectural structures of the processor, such as the branch predictor (more instances of the same branch might take longer to train) or the memory hierarchy (increased traffic on the instruction cache).

5.2 Object Allocation Context Correlation

5.2.1 Overview

In this section, we present a variation on the previous idea of correlating branch outcomes with the RAS by focusing our attention on programs built using object oriented patters. Such programs often make use of virtual calls, which are implemented in the underlying assembly as indirect jump instructions. We hence hypothesise that knowing the *place* where an object was allocated (i.e. the top element of the call stack) can help increase branch prediction accuracy within virtual functions.

5.2.2 Implementation

To evaluate this hypothesis, we augmented the captured execution trace with extra information about object allocations: whenever execution passes through a virtual function we require all trace entries generated within to be tagged with the topmost call stack item at the time of object allocation.

As such, Gem5 would need to detect object allocations and virtual method calls, and save the trace information accordingly. Gathering this data is not trivial though, as the compilation process discards most information about object allocations and types, hence making the problem difficult to solve within Gem5 alone. Inside the simulator, object allocations appear just like any other type of memory allocation, although with one key difference: objects are normally initialised via a (potentially inlined) constructor. In light of this, it becomes clear that a hybrid approach is needed here, in which the compiler emits more context towards the simulator.

Our solution to this problem centres around augmenting the Aarch64 ISA with a new, custom instruction, `obj_alloc_hint`, used for signalling object allocation. In this design, each constructor is instrumented with this new instruction such that later on Gem5 may decode it and record the object allocation. The main advantage of this design is its minimal impact on performance: while only introducing very little overhead to each constructor, it does not prevent the compiler optimisations such as inlining.

5.2.3 LLVM Compiler Pass

The LLVM Project comprises of a collection of tools and technologies for building compilers [13]. It is a mature project with very wide commercial use, and much of its success is due to its very modular structure. Compilers based on LLVM generally follow a simple structure, as in Figure 5.2. First, a compiler frontend parses code written in a high level language and generates the corresponding unoptimised LLVM Intermediate Representation (IR). Then, the LLVM middle end runs a number of passes and analyses over the IR in a long pipeline, producing at the end a highly optimised version of the IR. Finally, the optimised IR is sent to an ISA-specific back end to be transformed in executable code.

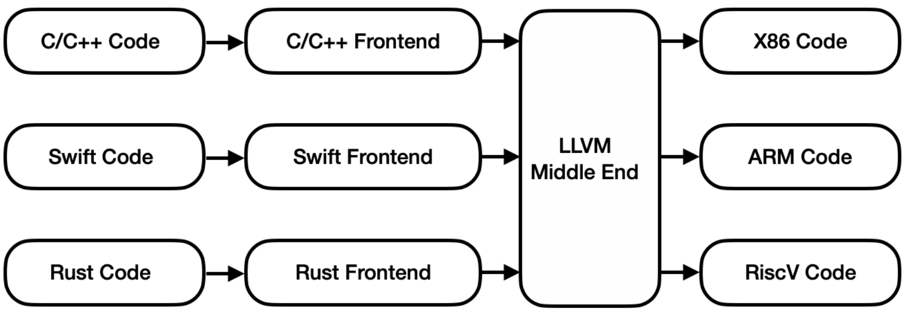


Figure 5.2: The Main Blocks of Different LLVM-based Compilers

For the purposes of object constructor instrumentation, a natural option to explore would be modifying the compiler front end. Since we are evaluating our work on SPEC 2006, whose object oriented benchmarks are written in C++, this would involve writing a plugin for the Clang front end. Although there is a compelling argument for this as Clang has access to the abstract syntax tree of the program and hence has full visibility of all user-defined constructors, this comes at the disadvantage that implicitly defined constructors are not captured.

In light of this, we decided to build an LLVM middle end pass to perform the instrumentation. Our pass is inserted in the *early extension point* of the simplification pipeline, i.e. it is run as early as possible in the middle end compilation processes such that further passes are free to perform optimisations on the instrumented code. Crucially, the pass is run prior to any form of inlining, which ensures that at this stage all object allocations in the IR call into the constructor.

Identifying constructors in the middle end is more challenging though. The LLVM IR is build to mimic the structure of assembly code and as such it does not capture the high level concept of a constructor; in effect, constructors appear just as regular functions. The solution to this problem becomes apparent when analysing the name of the symbols produced by Clang: as is the case with virtually all modern C++ compilers, the resulting code is built to follow the Itanium C++ Application Binary Interface (ABI) - an interface specification for binaries made to allow interoperability. We hence find that all constructors have their names mangled in a deliberate and specific way, with certain suffixes determining whether a function symbol is of interest to us:

- **Complete Object Constructors**: these constructors are invoked first upon object creation, and have the suffix "C1ev" under the ABI. To properly support inheritance and virtual methods, complete object constructors will set up the object's virtual table and may make further calls base class constructors.
- **Base Object Constructors**: these constructors are invoked when a derived type is instantiated, and have the suffix "C2ev" under the ABI.

The ABI defines two types of constructors, complete and base, with the purpose of aiding the implementation of virtual inheritance. For our problem though, extra care needs to be taken such that every object allocation is instrumented only once. Consequently, in our implementation, we begin by identifying all complete object constructors by demangling the function names in the IR and proceed to instrument them. This approach is sound because it correctly handles multiple inheritance: whereas multiple base constructors may be invoked whenever an object is created, only a single complete constructor will be called. As such, every object allocation is only logged once.

Performing the actual instrumentation is just a matter of adding the correct IR instructions to the beginning of each identified constructor. To avoid modifying the compiler back end, we emit the custom instruction by means of an inline assembly directive such that the correct instruction bytes are hard-coded in the assembly template. Special care was taken to instruct the compiler that the template has no side effect, although it does use the architectural register `x0` for reading. The code transformation is showcased in Figure 5.3. Although the instruction setting this register to the object address (line 7, Figure 5.3b) might look superfluous at first glance, it will play a very important role for data collection as it provides a way for the simulator to access the object address, regardless of any optimisations.

```
 1
 2
 3
 4  define ... void @_ZN1AC1Ev(ptr ... %0) ... {
 5    %2 = alloca ptr, align 8
 6    store ptr %0, ptr %2, align 8
 7    %3 = load ptr, ptr %2, align 8
 8    call void @_ZN1AC2Ev(ptr ... %3)
 9    ret void
10  }
11
12
13
14
```

(a) Before Instrumentation

```
 1  define ... void @_ZN1AC1Ev(ptr ... %0) ... {
 2    ; get object address from first parameter
 3    %2 = ptrtoint ptr %0 to i64
 4    ; inline assembly directive with
 5    ; hard-coded instruction bytes
 6    call void asm sideeffect
 7      "mov x0, $0\0A\09.long 0xFF550110",
 8      "r,~{x0}"(i64 %2)
 9    %3 = alloca ptr, align 8
10    store ptr %0, ptr %3, align 8
11    %4 = load ptr, ptr %3, align 8
12    call void @_ZN1AC2Ev(ptr ... %4)
13    ret void
14  }
```

(b) After Instrumentation

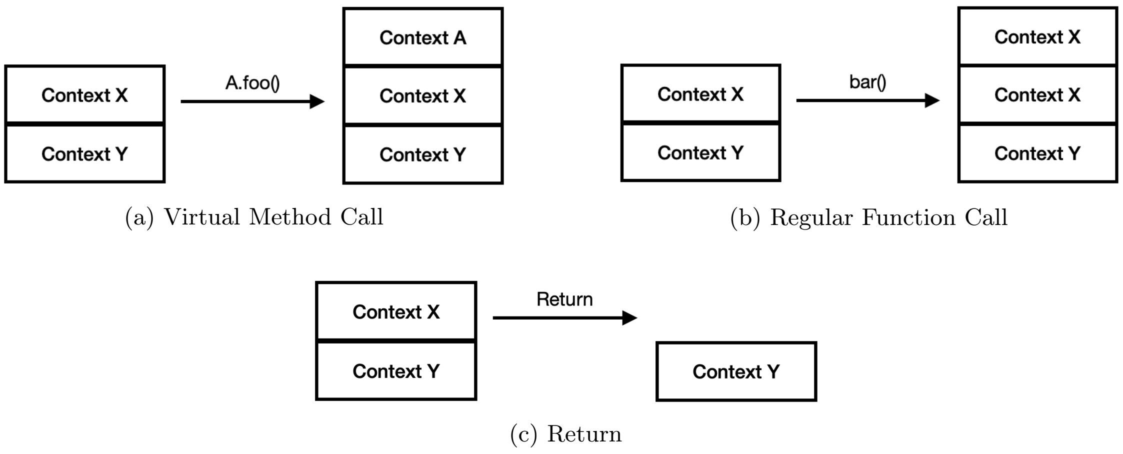
Figure 5.3: Simplified LLVM IR for a Sample Constructor

5.2.4 Gem5 Tracing

On the simulator side, we first needed to decide on how to encode the new *obj_ alloc_ hint* instruction. Fortunately, Gem5 already has support for "pseudo-instructions", which are custom additions to the ISA normally used for controlling the simulator from within the executing program. Gem5's pseudo-instructions use an unallocated instruction encoding that contains a control byte to select the function to be executed within the simulator. We therefore based our implementation on this infrastructure and augmented the Gem5 code base with support for a new value in the control byte, used for signalling an object allocation.

Building on this foundation, we augmented the branch tracer to record the address of every object upon its allocation. As detailed in the previous section, upon every object allocation the compiler will emit an instruction setting the `x0` register to the object address as well as the `obj_alloc_hint` instruction to signal the allocation to the simulator. As such, the simulator only needs to record in a table the value of `x0` and the top of the return address stack whenever the `obj_alloc_hint` is encountered.

The last step in this process is to tag each branch with its corresponding object allocation context. For indirect branches the process is rather simple, as upon calling a virtual method the compiler will always store the object address in the `x0` register. As such, Gem5 only needs to fetch this value and look it up the its table of allocation contexts. However, the process of tagging any other branch is not as clear, and hence allows for multiple interpretations. In this work, we take the view that all branches that happen after a virtual call are inherently linked to the call's context and as such will be tagged the same, even if they live in a different function (e.g. branches in another function called by the virtual method). As such, we implemented the tagging by maintaining a stack of contexts, where each branch gets tagged with the value on the top of the stack. Each virtual call pushes its own context to the stack, while regular function calls just duplicate the top most context. Finally, all function returns pop the context stack. The process is summarised in Figure 5.4.



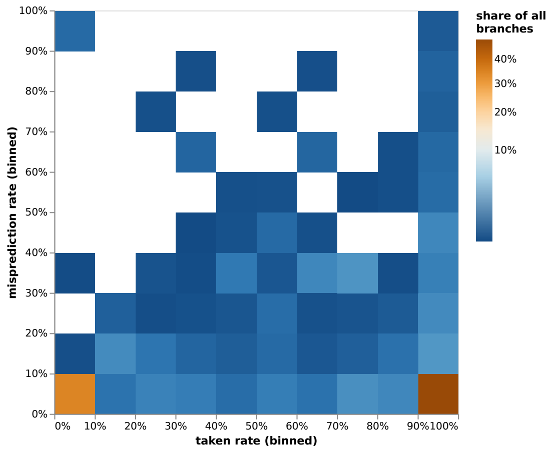
(a) Virtual Method Call (b) Regular Function Call (c) Return

Figure 5.4: Object Allocation Stack Maintenance

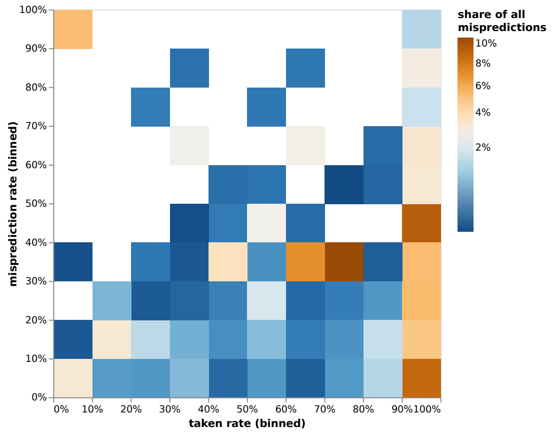
5.2.5 Case Study

Within this subsection, we will elaborate on the specifics of correlating the object allocation context with branch outcomes, as applied to the SPEC2006 483.xalancbmk benchmark on its training input. The benchmark is based on the Xalan C++ XSLT processor, which is a program that transforms XML documents into various other formats such as HTML. It consists of a very large code base that makes extended use of object oriented patterns, and as such serves as a great candidate for analysing our premises.

As in the previous section, we begin by estimating the global misprediction rate of the program through its SimPoint intervals, arriving at 4.25%. Following this, we compute the misprediction and taken ratios per branch, and aggregate the resulting data into bins to better visualise it. Results are presented in Figure 5.5. Figure 5.5a showcases the distribution of the unique branches in the program trace, where each bin groups together branches with similar taken and misprediction



(a) By Share of Occurrences



(b) By Share of Mispredictions

Figure 5.5: Binned Plots for the Branches in the SPEC2006 403.GCC Benchmark

rates. The colour of each bin is determined by the weight the branches carry with respect to their total occurrences in the trace. As such, bins with blue shades correspond to rare branches, whereas orange shades contain branches with high execution counts. As expected, most branches fall into bins on the bottom row, which correspond to low misprediction rates. Furthermore, many fall on the extremes of this row, which means they are highly unimodal (i.e. more than 90% taken or less than 10% taken, so they are easy to predict even for a local predictor).

Figure 5.5b is a variation on the same concept, where each bin's colour is determined by the share of mispredictions coming from the branches within. We observe multiple hot spots where mispredictions stem from. Firstly, it appears that many mispredictions happen for branches with takeness above 90%, for which even a local predictor should perform well. However, the same bins appear in the previous plot as having a low overall share of the total execution, so we could reason that these are just rare branches whose statistics are hard to learn due to sparse data. We also find another hot spot between 60% – 80% takeness and 30% – 40% misprediction rate. Since these numbers have a sum of around 100%, it appears likely that TAGE just learns to use a local predictor for these branches.

The above plots were built in rather coarse fashion as branches could only contribute to one bin. We can now introduce the main analysis for this section: we now create bins that aggregate branch/object allocation context pairs, with weights based again on their share of all mispredictions. This should provide us with some insight into the predictive power of the object allocation context. If it is indeed predictive, we should observe a change into the bin distribution when compared to the previous plot in Figure 5.5b, as each unique branch would now be able to contribute to multiple bins. Figure 5.6 illustrated the results of this experiment.

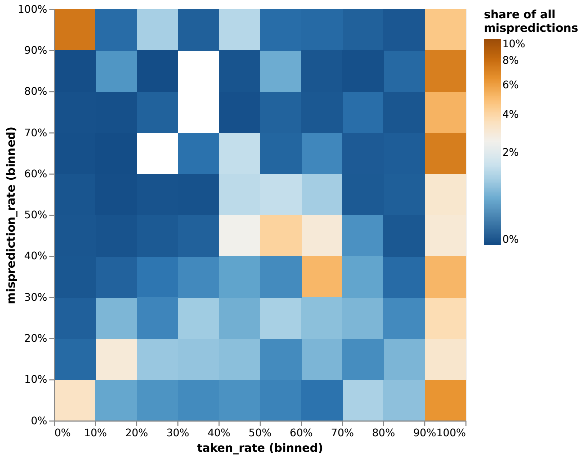


Figure 5.6: Binned Plot for Branch/Object Allocation Context Pairs

We indeed see a change in distribution, with branches being more evenly distributed across all bins. Furthermore, the hot spot at $60\% - 80\%$ takeness appears to have been greatly reduced, and we observe an increase in branches that are highly unimodal yet very poorly predicted (more than 90% misprediction rate with taken ration below 10% or above 90%). We believe this to be good evidence that future predictors could explore incorporating the object allocation context into their predictions.

Finally, as in the previous section, we provide an upper bound for the possible misprediction rate decrease when this approach is paired with local predictors. While not necessarily attainable in practice, it should nonetheless show the potential of the optimisation. To compute this, we calculate the takeness and misprediction rate for all branch/object allocation context pairs. Following this, we filter pairs based on whether applying a local predictor to them would improve performance as previously and then aggregate the results. For this benchmark, we obtain a maximum misprediction rate decrease of 2.44%, which in the ideal case would move the total misprediction rate to 1.81%.

5.3 Conclusion

To summarise, in this chapter we explored the effectiveness of introducing more context into branch prediction by collecting different pieces of microarchitectural state and correlating them with branch outcomes. We began by examining the predictive power of the microarchitectural return address stack, and presented some preliminary results of this idea in a case study based on a SPEC2006 benchmark based on the GNU Compiler Collection. Ultimately, we found that this idea only has marginal potential to improve branch prediction in this scenario. We also explored whether the call stack at the time of object allocation is predictive for branches executed in later method calls, with a higher degree of success, as we have found a good potential for improving performance. The implementation details of this approach were detailed alongside another case study into a SPEC2006 C++ benchmark based on the Xalan XSLT processor.

Chapter 6

Transformer-based Neural Branch Prediction

The focus of this chapter is to explore hidden correlations between the execution path of a program and the corresponding branch outcomes. Given the challenging nature of the problem, we hence approach it from a deep learning perspective and introduce a novel take on branch prediction inspired by the state of the art in the field of Natural Language Processing: treating branch prediction as a machine translation problem.

6.1 Motivation: From Taken History to Path History

Current state-of-the-art branch predictors (e.g. TAGE, Perceptrons) base their decisions on local branch statistics in the context of the recent taken branch history. It is an approach that has passed the test of time, being the dominant idea behind the best predictors of the last decades. In previous sections, we explored the effectiveness of extending the taken branch history context with extra information extracted from the return address stack either at the time of branch execution or, for object oriented languages, at the time of object creation. In this section though, we aim to provide some intuition into the shortcomings of the taken branch history context, and how using the full path history could alleviate these issues.

Consider, for the sake of argument, the example code listing below:

```
 1  if (condition A) {                        // B1
 2      ...
 3  }
 4  int N = random_int();
 5  for (int i = 0; i < N; i++) {
 6      if (random_boolean()) {               // B2
 7          ...
 8      }
 9  }
10  if (condition A && condition B) {         // B3
11      ...
12  }
```

Figure 6.1: Example of a Hard-To-Predict Branch (`B3`)

The code above is a synthetic example showcasing a hard-to-predict branch. Nevertheless, it has been chosen to reflect some of the findings of the researches at Intel as detailed in [1]. The code contains a branch on line 1 (`B1`) that is highly correlated with another branch on line 12 (`B3`). However, in between the two branches there is a loop that runs for an unknown and unpredictable number of iterations. Furthermore, the loop contains an unpredictable branch within it (`B2`), that will pollute the taken history with irrelevant entries.

This is a scenario in which both TAGE and Perceptrons fail. To understand why, consider the taken history (and hence the contents of the Global History Register) when predicting the branch `B3` (Figure 6.2; ignoring the branches needed to implement the `for` loop as they are not relevant for this argument).

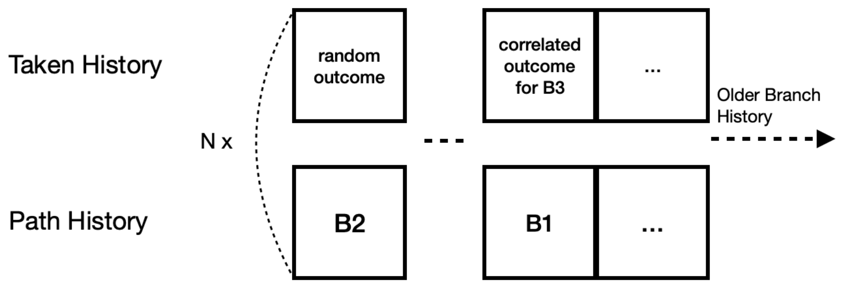


Figure 6.2: Branch History when Predicting Branch `B3`

As suggested by the diagram, the outcome of the correlated branch B1 appears at a random offset in the branch history. As such, a predictor based on the taken history alone (e.g. TAGE, Perceptrons) cannot learn this correlation because there is not enough information to differentiate between the entries in the GHR - from the point of view of the predictor, the GHR contains just random values.

In spite of this, the diagram suggests a possible solution: a mechanism is needed that would use the path history to *filter the branch history*, or, in other words, a method to instruct the branch predictor to which history positions it should *attend* to. It is under this light that we developed a Transformer-based neural network that performs precisely this function.

6.2 Transformer Nets and Sequence-to-Sequence Models

Introduced by Vaswani et. al in 2017, the Transformer architecture [14] has started a true revolution in the field of Natural Language Processing (NLP). Particularly well suited for sequence-to-sequence problems such as machine translation, it has greatly surpassed the performance of previous models. In this section, we will give a high-level overview into the inner workings of the Transformer Architecture as well as cover some of its advantages and how these relate to previous models.

6.2.1 Sequence-to-Sequence Models

As presented in the original paper, Transformers fall in the larger family of sequence-to-sequence models. These are models designed to solve problems in which a source sequence needs to be converted into a target sequence, potentially in a different language and even with a different meaning. Common examples in NLP include machine translation (e.g. translating texts form Romanian to English), document summarisation or chat bots.

A common pattern in the design of sequence-to-sequence models is the usage of the *encoder-decoder* architecture, where the model is split between two functional blocks. Firstly, an encoder block is tasked with analysing the source sequence so that it may represent its semantics into some underlying embedded space. The results of the encoder are then passed to the decoder so that it may then generate the target sequence.

6.2.2 Token Embeddings

As is the case with virtually all other machine learning algorithms operating on sequences, Transformers do not operate on tokens directly, but rather on their embeddings. Simply put, token embeddings are just numerical representations of the tokens in a high dimensional vector space. Each token has its own unique embedding vector, whose entries are model parameters that can be learned through gradient descent just like any other internal weight.

6.2.3 Recurrent Neural Networks

Prior to the introduction of the Transformer, some of the most successful approaches for tackling problems with sequential data were the ones based on Recurrent Neural Networks (RNNs). The core idea behind such networks is to repeatedly apply some functional unit (i.e. a neural network *cell*) to each token of an input sequence, while passing some fixed-sized embeddings from one cell to the next to maintain some memory of past tokens, as illustrated in Figure 6.3. As a technicality of the learning process and gradient descent, this approach is particularly prone to exploding/vanishing gradients; as such the academic community proposed alternative cell models that aim to alleviate this issue in the form of LSTM [15], GRU [16] cells.

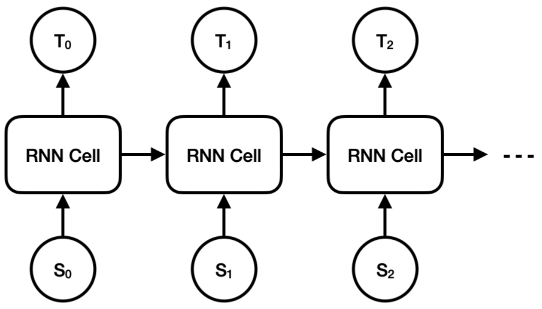


Figure 6.3: RNN Cells

RNNs are still commonly deployed to solve sequence-to-sequence tasks, often using an encoder-decoder topology, as the token alignment and length may differ from the source to the target sequence. As such, the decoder is generally built to be *auto-regressive*, i.e. the output of one decoder cell is sent as the input to the next, alongside the memory.

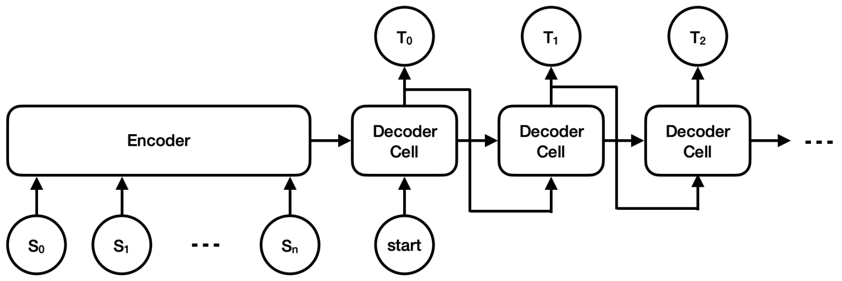


Figure 6.4: Encoder-Decoder RNN with Auto-Regressive Decoder

Although they have been widely deployed to many tasks in NLP with good success, their design poses a fundamental information bottleneck in the fixed-size memory embedding, which consequently limited their ability to learn very long correlations between the tokens of a sequence.

6.2.4 Attention Mechanism

The Transformer architecture addresses the limitations of prior methods by employing the *Scaled Dot-Product Attention* mechanism as a primary way of capturing correlations between the tokens of a sequence. In essence, the mechanism aims to reduce the *distance* that information has to travel from one token to any other.

Intuitively, the Attention mechanism has many similarities to search algorithms. To begin with, the algorithm takes as input a sequence of embeddings representing keys K, queries Q and values V. Under the search interpretation, the keys can be viewed as containing some internal representation of the meaning of the sequence tokens, the queries would be filters used for finding other related tokens in the sequence and the values would contain some extracted information about the tokens. The algorithm then matches all queries with all keys such that it would obtain an attention score for each pair of tokens; the higher the overlap between the query and the key, the more the tokens are deemed to be correlated. The attention weights are then used in a weighted sum between the value embeddings, resulting in new, context-aware embeddings for each token.

Formally, as per the original paper [14], the embeddings in K, Q and V are taken to have the same dimension d_k. The attention weights are obtained by performing the dot product between the query and key vectors, followed by a scaling of $\frac{1}{\sqrt{d_k}}$ and the softmax activation function (Figure 6.5). The weights can then be used to create a weighted average of the V embeddings. The original paper also introduces the concepts of self-attention, where the same embeddings are chosen for the keys, queries and values and multi-head attention, where attention is applied multiple times to different projections of the input sequences, thus forming multiple heads whose results are then concatenated and projected to form the final output.

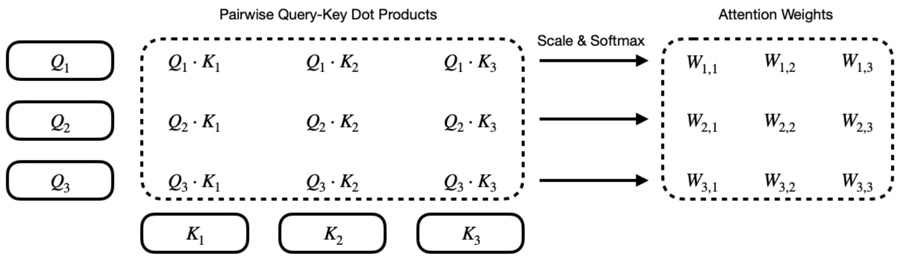


Figure 6.5: Computing Attention Weights for a Sequence of Three Elements

The complete process is succinctly summarised by the following equations, taken as an extract from the original paper:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V$$

$$\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$

6.2.5 Transformer Networks

As is the case for other sequence-to-sequence models, Transformer networks follow an encoder-decoder architecture as well. The Transformer Encoder is built by stacking a number encoder layers, each of whom contains a multi-head self-attention block and a point-wise fully-connected layer. The output of the encoder, also called *memory*, will be used as part of the decoder. The decoder is built in a similar manner, though in addition each decoder layer contains a cross-attention block, that takes its queries from previous layers, but the keys and values from the encoder memory. Residual connections and normalisation layers are also added throughout the network to improve training.

Nevertheless, making proper use of this network architecture requires special care for a number of subtle details regarding its inner workings. Firstly, as a consequence to the pairwise matching of tokens in the input sequence followed by a weighted sum, the self-attention block is permutation equivariant. Naturally, this is an important issue, as in most sequence-based problems the order of the tokens is highly relevant. The solution to this problem is to feed the network *positional encodings*: the sum of regular token embeddings with some positional information, which can either be fixed or learned.

Finally, there is another subtle issue is with regards to the way the attention weights are computed. As stated earlier, the network computes pairwise matches between all queries and all keys to computer the weights. This is not always desirable as in many problems tokens should not be able to peek/attend at future tokens. As such, masking has to be applied, so as to prohibit the attention mechanism to assign weights to future tokens.

In Figure 6.6, we present an diagram based on the original paper on Transformers which illustrates the complete model architecture.

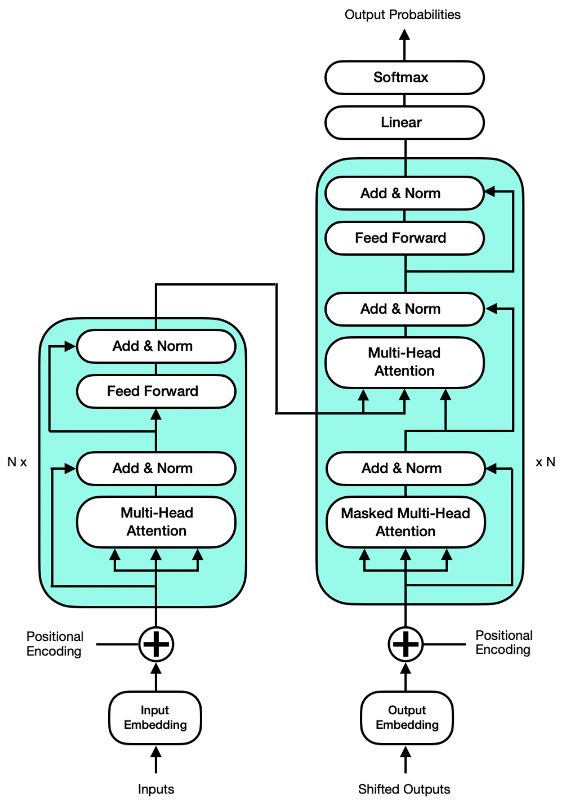


Figure 6.6: The Transformer Architecture (based on a diagram from [14])

6.3 Branch Prediction as a Machine Translation Problem

Considering the limitations of predictors based on the taken history as presented earlier, we propose a Transformer-based branch predictor that operates on a combination between the path history and the take history. The model is based on a novel perspective on branch prediction, that it can be viewed as a machine translation problem from the domain of path history sequences to the domain taken history sequences.

To begin, as is the custom in NLP, we will formalise the alphabet of our problem. The input tokens are formed of branch addresses, and are sent to the encoder in the sequence in which they have been encountered, i.e. the recent path history. To reduce the amount of source alphabet tokens, we may also apply a mask to the address (e.g. to keep only the low order bits). The decoder operates on the branch outcomes, and as such the target sequence alphabet is made out of the `"taken"` and `"not taken"` tokens. A further token is added to the output alphabet, namely the `"start of sequence"` token, so as to indicate to the decoder that it should start predicting branch outcomes.

For the purposes of this work, we will employ learned positional encodings, i.e. the learned token embedding vectors are to be summed with corresponding learned embedding vectors for their position in the sequence before being sent to the Transformer network (Figure 6.7). As such, the maximum history length supported by the model becomes a hyper-parameter.

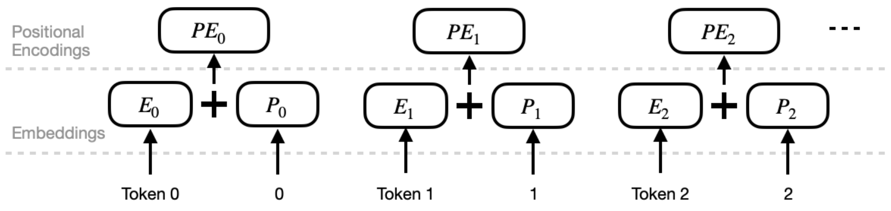


Figure 6.7: Calculating Positional Encodings

To produce a branch prediction out of the Transformer network, the encoder has to be provided with the recent history of branch addresses (including the current candidate branch for which a prediction will be made as the last token). The decoder has to be provided with a shifted sequence of recent branch outcomes - shifted in the sense that the sequence has to start with the `"start of sequence"` token and, of course, does not contain the outcome for the candidate branch (i.e. the last branch in the source sequence). For each input token, the decoder will hence produce a prediction, as illustrated in Figure 6.8. Therefore, the last decoder output will correspond to the branch prediction for the candidate branch.

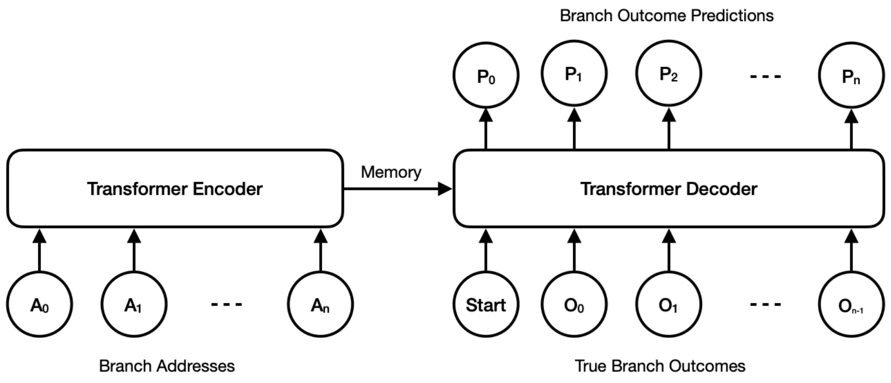


Figure 6.8: Predicting Branch Outcomes with the Transformer Model
$(A_i)_{i=0}^{n}$ is the sequence of branch addresses in the path history, $(O_i)_{i=0}^{n-1}$ it the sequence of true branch outcomes and $(P_i)_{i=0}^{n}$ is the sequence of branch predictions. The candidate branch A_n is predicted as having the outcome of P_n.

6.4 Training Transformer-based Branch Predictors

When training a Transformer model such as the one proposed above, special care has to be taken to a number of subtle details. In this section, we will describe in detail the training procedure as well as the rationale behind the different choices we made while designing it.

6.4.1 Parallelising Gradient Outflows for Efficient Training

The prediction method described in Figure 6.8 has in effect a single output, i.e. the outcome for the candidate branch. The other outputs could be considered superfluous, as the true branch outcomes are already known for those positions. Still, they will play a crucial role in the training of the model, as we will now detail.

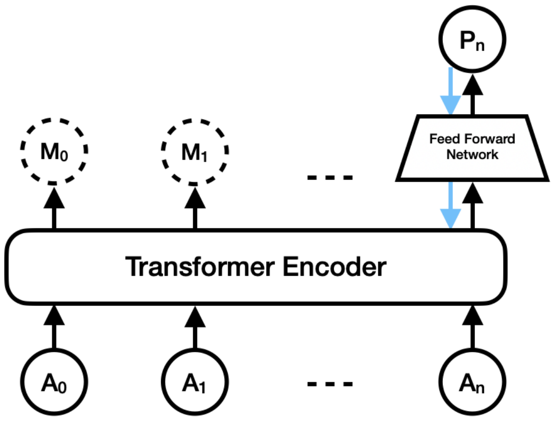


Figure 6.9: Poor Gradient Flow with One Output
Blue arrows represent gradient flows. Dashed lines indicate ignored memory outputs.

Initially, while designing the model, we experimented with using simpler training procedures as well as simpler models. We experimented with using only a Transformer encoder with outputs derived through a feed forward network and a Sigmoid activation from the last memory entry. As such, the for every input combination, we would compute the Binary Cross Entropy loss for this single output and perform gradient descent on the result. Unfortunately, this model was not able to converge or learn any branch behaviours, achieving an accuracy of near 50% on out benchmarks. We believe this to be due to inefficient training, in particular due to poor gradient flows through the network. To further elaborate, consider Figure 6.9. The diagram shows how for this design, the network only learns about the output of one branch at a single time, and only with a fixed history length. Moreover, intermediate memory outputs have no gradient inflow or outflow, so part of the network does not get gradient updates.

The solution to this problem is to perform training in parallel for all output tokens. Apart from the benefit of having good gradients (Figure 6.10), this method also has the advantage of guiding the network to learn something about the behaviour of each branch in the training sequence in parallel, as well as exposing the network to different sequence lengths (e.g., the first branch will observe no history, the second one will only observe the previous branch etc.).

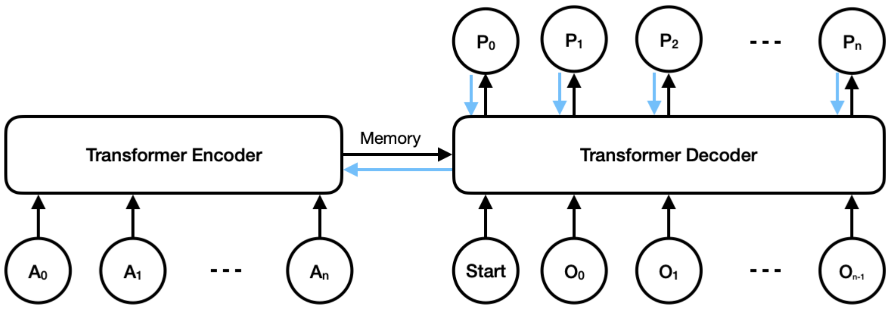


Figure 6.10: Good Gradient Flow with Multiple Outputs

6.4.2 Attention Masking

Training a model with multiple outputs as presented in previous sections imposes a number of restrictions on the internal attention weights generated by the model. Intuitively, we want to ensure that the prediction made at each position is based solely on information resulting from preceding tokens in the sequence, as this is how the network will be used in practice. Consequently, we will have to apply attention masks to the different attention blocks in the network.

Starting with the encoder, a triangular mask is applied to the self-attention block such that entries to the right of any sequence position are ignored. This has the effect of making every resulting memory cell only contain information up to its position, i.e. when making predictions, the network is not allowed to look into the future of the path history. Similarly, the decoder self-attention will be masked with an identical mask so as to prohibit looking into the future of the taken history. Finally, the decoder cross-attention block requires the same mask to be applied, as not doing so would risk leaking information about the future path history through the memory embeddings.

6.4.3 SimPointed Datasets and Weighted Loss Function

An early, yet highly important step in training any machine learning model is gathering good datasets for training, validation and testing. At first glance, this may appear as a simple endeavour: choose a target benchmark program and run it with different inputs in a simulator such as Gem5, while capturing as in previous chapters an execution trace containing the complete path and taken histories. Such a dataset should be highly representative for the kinds of branch histories exhibited by the program. Unfortunately, this simple approach comes with a very significant drawback: most real programs run in excess of even billions of instructions and as such would yield immense training datasets. Furthermore, transformers are notorious for being large models that are slow to train. It hence becomes clear that this training procedure quickly becomes impractical.

Nevertheless, we aim to tackle this challenge by first hypothesising that large parts of these potential execution traces would be in fact redundant. The success of SimPoint does appear to suggest that this is the case, and as such we propose that training is to be done solely on traces captures in intervals selected by SimPoint (as elaborated in Section 4.2). Naturally, we cannot simply aggregate the interval traces into a normal dataset as this would capture wrong statistics about the program: different intervals (and hence traces) hold a different share of the total program execution. In much the same way we weighted statistics based on the SimPoint weights in previous chapters, we now aim to weight the resulting gradients of the training procedure. We achieve this by using a weighted Binary Cross Entropy loss function, where each training entry is weighted by its corresponding SimPoint interval weight. The final formula for the loss function is as follows (where N is the batch size, p is a vector of predictions, t is a vector of targets, and w is the corresponding vector of SimPoint weights. Note that the output sequences are flattened for computing the loss):

$$\text{WeightedBCE}(p, t, w) = -\frac{1}{N}\sum_{i=1}^{N}\left[t^{(i)}\log(p^{(i)}) + (1-t^{(i)})\log(1-p^{(i)})\right]\cdot w^{(i)}$$

6.5 Case Studies

In this section, we will present a number of case studies showcasing the power of the Transformer architecture when applied to branch prediction.

6.5.1 Synthetic Benchmark

We begin by analysing a synthetic benchmark that we have designed by hand to include a hard-to-predict branch. A simplified version of the benchmark is presented in Figure 6.11. The program is loosely based on the example code in Section 6.1, and contains two correlated branches, `B1` on line 5 and `B3` on line 15. In between them we can find `B2` on line 10, which is an uncorrelated branch with unpredictable behaviour that pollutes the branch history. We aim to gain insight into whether the Transformer can discover that `B2` is not correlated.

```
 1  int acc = 0;
 2  for (int i = 0; i < ITER; i++) {
 3
 4      int long_dependency = randint();
 5      if (long_dependency % 2 == 0) {
 6          long_dependency += 2;
 7      }
 8      int n = randint() % 16;
 9      for (int j = 0; j < n; j++) {
10          if (randint() % 2 == 0) {
11              long_dependency += 4;
12          }
13      }
14
15      if (long_dependency % 2 == 0) {
16          acc+=8;
17      }
18  }
```

Figure 6.11: Caption

As the benchmark is very simplistic and fast to execute, we built a training dataset for it without SimPoint. We hence trained a Transformer model on the resulting trace, with the following hyper-parameters:

- Maximum sequence length of 64 tokens
- 2 encoder layers and 2 decoder layers
- 2 head Multi-head attention
- Embedding dimension of 512

- Internal Feed Forward layers of 128 neurons
- Adam Optimiser with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 1.0e-9$ and learning rate of 0.0001
- Batch size of 32
- 5 Epochs

The model quickly achieves more then 95% accuracy on the train set. To examine whether the model successfully learned that branch `B2` is not correlated, we analyse the attention weights computed in an arbitrary window of a trace from another run.

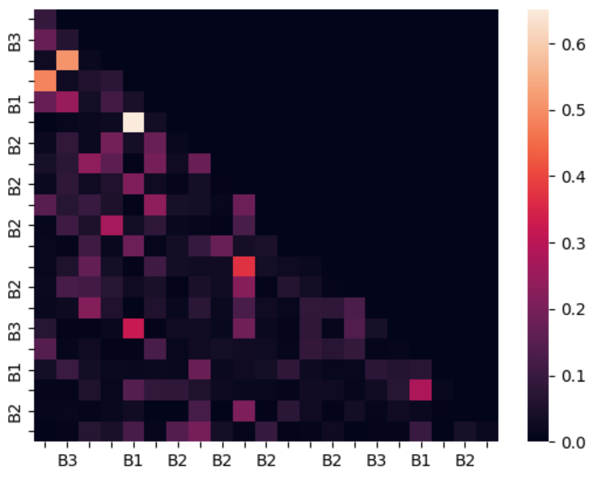


Figure 6.12: Slice of the Attention Weight Matrix for the Encoder Layer 0 (average of the two attention heads)

Figure 6.12 showcases a relevant slice of the resulting attention matrix as a heat map. Each row represents the attention weights that were computed for a particular occurrence of a branch in the path history. Hence, by fixing a row, we can find the computed attention for each position in the sequence by analysing the columns. Branches are listed in order of their occurrence, from top to bottom and from left to right. Given that tokens are not allowed to attend to future positions, the plot takes on a triangular shape. It also contains a number of bright spots which we will suggestively call *fireflies* - these correspond to positions of high attention.

We observe that branch `B3` appears twice in the sequence. The first occurrence is not very relevant, because it occurs early in the sequence and thus does not have access to a long history. However, the second occurrence exhibits a firefly, precisely for the position corresponding to the most recent occurrence of `B1`. Moreover, it appears as though the network learned to ignore positions corresponding to `B2`, as they have very low attention scores (and hence appear dark in the plot). We believe these results give strong circumstantial evidence that the model is working as intended.

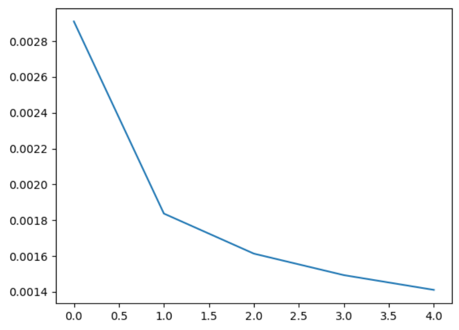
6.5.2 445.gobmk

We also trained and validated a model on the SPEC2006 445.gobmk benchmark. This benchmark is based on an AI program that plays the well-known game of Go.

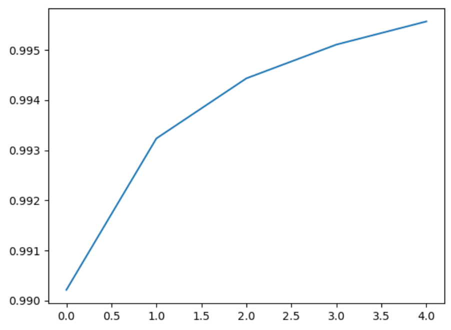
To build a training dataset for this benchmark, we ran all the *train* benchmark inputs under Gem5, calculated simpoints for each run and proceeded to build execution traces for each SimPoint interval as detailed in Chapter 4. The final dataset was an aggregate of all traces alongside their respective SimPoint weights (although not necessary, we also re-normalised the weights by dividing them by the number of benchmark inputs). The validation set was built based on traces from the first 10,000,000 instructions of each *ref* input set to allow for faster evaluation. Based on validation results, we tuned the hyper-parameters of the network by hand and arrived at the following network parameters:

- Maximum sequence length of 512 tokens
- 8 encoder layers and 8 decoder layers
- 8 head Multi-head attention
- Embedding dimension of 512
- Internal Feed Forward layers of 128 neurons
- Adam Optimiser with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 1.0e - 9$ and learning rate of 0.0001
- Batch size of 32
- 5 Epochs

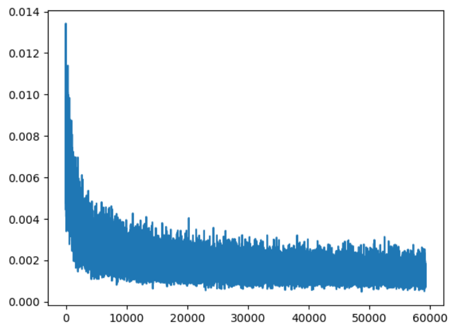
Below are the loss and accuracy curves for the train and validation sets:



(a) Average Train Loss per Epoch



(b) Validation Accuracy per Epoch



(c) Train Loss per Batch

Figure 6.13: Loss Curves for Training on the SPEC2006 445.gobmk Benchmark

Finally, we also took another slice of an attention weight matrix generated for some arbitrary window in the training set. As can be seen in Figure 6.14, fireflies are present once again, indicating learned correlations of the program branches.

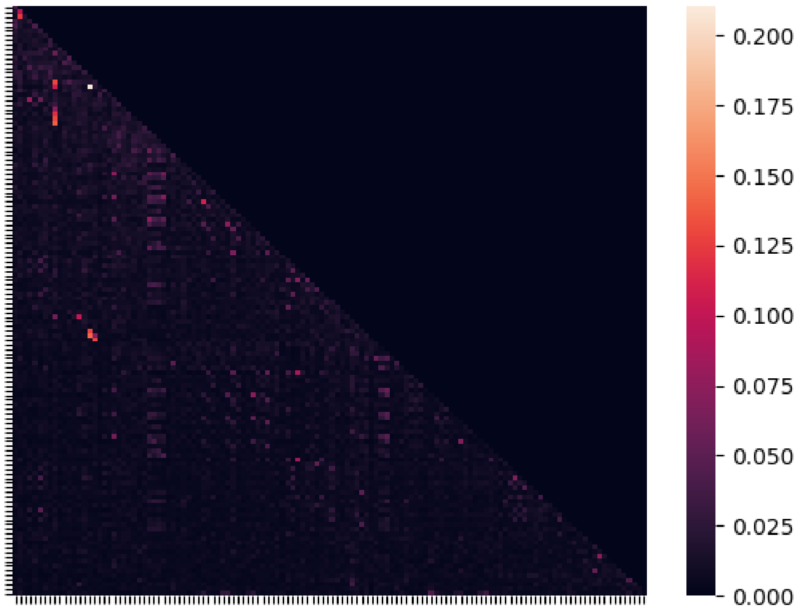


Figure 6.14: Attention Matrix in the SPEC2006 445.gobmk benchmark

6.6 Summary

To conclude, in this chapter we introduced a novel approach on branch prediction inspired by Natural Language Processing and Machine Translation. We applied Transformer networks to solve the problem, and detailed a training procedure that is fast and efficient based on SimPoints. We also explored a number of case studies, and showed evidence that suggests that the networks trained with this approach are able to learn subtle dependencies between branches.

Chapter 7

Evaluation

In this chapter, we will provide a formal evaluation of the methods described so far in this work on the SPEC2006 integer benchmarks.

7.1 Methodology

All results in this chapter will be provided as a weighted aggregate over a number of SimPoint intervals extracted from the *test* inputs of each benchmark. The data generation process will follow the same procedures as described in Chapter 4. For benchmarks containing more than one set of inputs, we will further average the metrics over the different inputs.

7.2 Return Address Stack Correlation

Benchmark	TAGE-SC-L 64KB Misprediction Rate	RAS Correlation Opportunity (Upper Bound)
400.perlbench	5.9%	2.7%
401.bzip2	5.9%	0.1%
403.gcc	3.2%	0.8%
429.mcf	9.6%	0.6%
445.gobmk	7.9%	0.7%
456.hmmer	5.1%	0.5%
458.sjeng	8.4%	0.8%
462.libquantum	3.4%	0.1%
464.h264ref	4.4%	0.2%
471.omnetpp	2.3%	1.1%
473.astar	16.0%	0.5%
483.xalancbmk	1.7%	0.9%
Average	6.1%	0.7%

Table 7.1: Optimisation Opportunities Across the SPEC2006 Integer Benchmarks when Correlating the Top Element of the Return Address Stack

In Table 7.1, we present the final evaluation across the full SPEC2006 Integer benchmark suite. We find an overall average opportunity to reduce the misprediction rate by 0.7%. The results show that within the benchmark suite, there exists small fraction of branches that are in fact correlated to a degree with the return address stack. However, the results represent only an upper bound, and in all likely hood in practice the observed benefit would be smaller.

7.3 Object Allocation Context Correlation

Benchmark	TAGE-SC-L 64KB Misprediction Rate	Object Allocation Context Correlation Opportunity (Upper Bound)
471.omnetpp	2.3%	0.9%
473.astar	16.0%	0.0%
483.xalancbmk	1.7%	0.7%
Average	6.6%	0.5%

Table 7.2: Optimisation Opportunities Across the SPEC2006 Integer Benchmarks when Correlating the Top Element of the Return Address Stack at the time of Object Allocaion with Branches in Virtual Methods

In Table 7.2 we present the results on the 3 C++ Integer benchmarks of the SPEC2006 suite. We can observe once again modest prospects for optimisations, and in one specific case, i.e. the `astar` benchmark, no opportunity at all.

7.4 Transformer-based Neural Branch Prediction

	TAGE-SC-L 64KB Accuracy	Transformer Model Accuracy		Delta Accuracy	
		Known Branches	All Branches	Known Branches	All Branches
400.perlbench	94.1%	85.6%	83.0%	-8.5%	-11.1%
401.bzip2	94.1%	96.8%	94.9%	+2.7%	+0.8%
403.gcc	96.8%	94.7%	91.3%	-2.1%	-5.5%
429.mcf	90.4%	94.7%	94.5%	+4.7%	+4.1%
445.gobmk	92.1%	94.5%	94.1%	+2.4%	+2.0%
456.hmmer	94.9%	90.3%	90.4%	-4.6%	-4.5%
458.sjeng	91.6%	91.1%	91.0%	-0.5%	-0.6%
462.libquantum	96.6%	97.2%	94.6%	+0.6%	-2.0%
464.h264ref	95.6%	97.8%	97.7%	+2.2%	+2.1%
471.omnetpp	97.7%	95.6%	82.8%	-2.1%	-14.9%
473.astar	84.0%	91.1%	90.9%	+7.1%	+6.9%
483.xalancbmk	98.3%	90.0%	87.8%	-8.3%	-10.5%
Average	93.8%	93.3%	91.1%	-0.5%	-2.7%

Table 7.3: Accuracy obtained by the Transformer model on the SPEC2006 Integer Benchmark Suite

Table 7.3 showcases the final results after training a Transformer network on the SPEC2006 *train* benchmark inputs and evaluating on the *test* benchmark inputs inputs. The network architecture was chosen to be the same as in Subsection 6.5.2. For clarity, we will list the hyperparameters here once more:

- Maximum sequence length of 512 tokens
- 8 encoder layers and 8 decoder layers
- 8 head Multi-head attention
- Embedding dimension of 512
- Internal Feed Forward layers of 128 neurons
- Adam Optimiser with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 1.0e - 9$ and learning rate of 0.0001

- Batch size of 32
- 5 Epochs

Given that the Transformer network is an offline algorithm (i.e. no training is done during inference), we differentiate in the table between the *known branches* and *all branches*. While testing, we have found that the SimPoint intervals we used for training did not cover all branches in the program. As such, when reporting results we considered unknown branches as being precisely 50% correctly predicted (i.e. the accuracy of a coin flip) and aggregated results in the *All Branches* columns. We also reported the accuracy for branches encountered in training in the *Known Branches* columns.

7.4.1 Analysis

When analysing the results above, one has to be careful in comparing the two predictors as they did not have a level playing field. On the one hand, TAGE is an online algorithm and as such could learn about the statistics of the program while it was executing. At the same time, the Transformer might have suffered from covariate shift, i.e. the test distribution was not the same as the one it was trained on. On the other hand, the Transformer network is a much larger mechanism than TAGE, and was also trained on much more data, which could have given it an advantage when testing.

Therefore, the two predictors cannot be put clearly in a hierarchy. However, we believe that the results achieved by the Transformer network are very promising and could warrant further research into optimising the architecture: the model achieved comparable performance to TAGE, and even surpassed it in certain benchmarks. Furthermore, although not necessarily implementable in hardware in its current form, the Transformer model provides a large set of weights that could be analysed to discover hidden branch correlations, similar to how we analysed the self-attention weights in earlier sections.

Chapter 8

Ethical Considerations

Branch prediction is a fundamental aspect of virtually all modern processors. Although often overlooked, it has an influence whenever computers are used, which can be both for good and for malice. On the one hand, the performance increases brought by this technology can be used for the greater good of humanity through ethical research activities or simply by helping people be more productive in their daily live. On the other hand though, greater computing power can be used for disreputable purposes as well such as weapon development.

As such, its use is a double edged sword: at the same time, it can be of use for both good and bad actors. In recent times, we have also seen an increase in cybersecurity attacks specifically targeting the branch predictor such that sensitive information may be exfiltrated through micro-architectural side-channels. The attacks, collectively known as Spectre and Meltdown, affect to some degree all high performance processors. Various attempts to reduce their impact have been made, but thus far no comprehensive solution has been found, apart from removing branch prediction altogether - a compromise in performance which is impractical to make.

Our work in this research project neither solves nor aggravates these ethical issues. On the whole, branch prediction is a fundamental technology and the success of modern processors has also been due to its development.

However, we would like to address another issue in the greater field of computer architecture: the problem of open research. It is often the case that some of the greatest developments in the field are published long after their discovery or even never at all. As such, their potential impact on shaping the research literature is often diminished. Commercial corporations that design microprocessors have a natural incentive to limit the amount of information shared as their businesses are often built around a competitive performance advantage. The academic world sometimes suffers from this issue as well: although research papers are normally openly available through established journals, numerous ones do not fully publish their designs and limit themselves to high level descriptions. More often than not, new ideas in computer architecture are experimentally evaluated, hence making concrete designs and evaluation environment an invaluable resource for further research. Consequently, we intend to promote transparency and accessibility in our research by making our work open source.

Finally, our planned evaluation methodology is based around the SPEC 2006 benchmark suite. The software is protected by copyright and licensed under the *SPEC GENERAL LICENSE AGREEMENT* license. We have used the benchmark suite as part of this work by virtue of the license given to the project supervisor.

Chapter 9

Conclusion

9.1 Summary of Results

Overall, the aim of this project has been to explore and find new avenues for incorporating more context into branch prediction. We believe we have been successful in this search, as this work introduced a number of potential context sources for future branch predictor, alongside an evaluation over potential performance gains. To be specific, the most important contributions are:

- A study into the correlation opportunity between the Return Address Stack and branch outcomes. We have found that on average, the upper bound for the achievable misprediction rate decrease on the SPEC2006 Integer Benchmarks is 0.7%, i.e. on average from a misprediction rate of 6.1% to 5.4%, so around 12% net reduction in misprediction rate.
- A study into the correlation opportunity between the Return Address Stack at the time of object allocation and branch outcomes within virtual functions. We have found that on average, the upper bound for the achievable misprediction rate decrease on the SPEC2006 Integer Benchmarks is 0.5%, i.e. on average from a misprediction rate of 6.6% to 6.1%, so around 8% net reduction in misprediction rate.
- A novel perspective on branch prediction as a Machine Translation problem, and the introduction of a Transformer-based neural network designed for branch prediction.
- An efficient training procedure based on SimPoints for training neural branch predictors
- An evaluation into the effectiveness of using Transformer Neural Networks for branch prediction. On average, the model performed within 0.5% accuracy on branches encountered in training when compared to TAGE-SC-L 64KB.

9.2 Limitations

As is the case with any academic work, the results presented in this paper are subject to the scrutiny of the wider academic community. In this section, we aim to provide a list of known limitations of this work, such that further research based on it can address them or take them into account.

- **SimPoints**: using SimPoints has both advantages and disadvantages. One the one hand, they can greatly increase the speed of reaseach. On the other hand, they are only approximations to the full program execution, and, as we have seen when training the Transformer model, they do not have 100% coverage of the code.
- **Warm up Length**: When resuming simulation checkpoints, we use a $1,000,000$ instruction interval for warming up the internal micro-architectural structures of the processor (e.g. caches, BPU). However, we have not run experiments to prove that this warm up length is appropriate.

- **Rare Branch Statistics**: to evaluate the Return Address Stack correlations and the Object Allocation Context correlations we computed an upper bound for the possible improvement to the misprediction rate. We did so by considering every branch in the trace, and the calculating whether a local predictor paired with the context would yield better predictors. However, this approach is prone to outliers in the form of rare branches, which could skew the statistic, as their rare occurrences may not be representative of their actual behaviour.
- **Transformer Hyper-parameter Tuning**: We tuned the Transformer model by hand on a single benchmark. Given enough compute power, the ideal procedure would be to employ a form of automated hyper-parameter turning (e.g. grid search) and perform the tuning individually for every benchmark.
- **Over-fitting to the Benchmark**: 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 a number of possible ideas and research directions that are beyond the scope of this project, but which we nonetheless believe as having high potential for great results.

9.3.1 Branch Sets

A future possible research idea is that of *Branch Sets*. While building the Transformer model, we identified that a lot of issues in branch prediction could be solved by properly filtering the taken history and removing irrelevant entries. As such we imagined grouping the branches in a program into sets. The ISA would expose a number of architectural sets, and all branch outcomes from the members of a particular set would be pushed into a set-specific GHR. Furthermore, when making a branch prediction, only the branch's set GHR would be used to index into the BPU. Ideally, correlated branches would be assigned to the same set, such that the branch predictor can make effective use of the branch history. For example, the set assignment could be computed before run time by analysing the weights learned by a Transformer network.

9.3.2 Hardware Attention

Another research direction would be that of exploring hardware implementations for the attention mechanism. In the past, Courbariaux et al. have explored the concept of binarized neural networks [17]: neural nets with weights and activations that only take the values -1 and 1. Such networks can be effectively implemented in hardware, though have lower performance and can be hard to train. Future research could be centred around improving these techniques so that hardware attention mechanisms become viable.

9.3.3 Binary Instrumentation Trace Generation

Currently, we generate traces for training Transformer networks by simulating programs in Gem5. Although this approach has many advantages for research as it provides easy access to micro-architectural state, it can be rather slow in practice. However, for the purpouses of training a Transformer network only the path history and taken history are required. This information could be collected by a plugin for a binary instrumentation tool such as Intel Pin [18].

List of Figures

Bibliography

[1] Lin CK, Tarsa SJ. Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions. In: 2019 IEEE International Symposium on Workload Characterization (IISWC); 2019. p. 228-38.

[2] Seznec A, Michaud P. A case for (partially) TAgged GEometric history length branch prediction. The Journal of Instruction-Level Parallelism. 2006;8:23.

[3] Zhao J, Korpan B, Gonzalez A, Asanovic K. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. Fourth Workshop on Computer Architecture Research with RISC-V. 2020 May.

[4] Hornik K, Stinchcombe M, White H. Multilayer Feedforward Networks Are Universal Approximators. Neural Netw. 1989 jul;2(5):359–366.

[5] Jimenez DA, Lin C. Dynamic branch prediction with perceptrons. In: Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture; 2001. p. 197-206.

[6] Corporation A. AMD; 2016. Available from: https://www.anandtech.com/Gallery/Album/5197#18.

[7] Fog A. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering. 2022;3.

[8] Seznec A. Tage-sc-l branch predictors. In: JILP-Championship Branch Prediction; 2014. .

[9] Chen ICK, Coffey JT, Mudge TN. Analysis of Branch Prediction via Data Compression. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS VII. New York, NY, USA: Association for Computing Machinery; 1996. p. 128–137. Available from: https://doi.org/10.1145/237090.237171.

[10] Krall A. Improving Semi-Static Branch Prediction by Code Replication. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation. PLDI '94. New York, NY, USA: Association for Computing Machinery; 1994. p. 97–106. Available from: https://doi.org/10.1145/178243.178252.

[11] Young C, Smith MD. Improving the Accuracy of Static Branch Prediction Using Branch Correlation. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS VI. New York, NY, USA: Association for Computing Machinery; 1994. p. 232–241. Available from: https://doi.org/10.1145/195473.195549.

[12] Hamerly G, Perelman E, Lau J, Calder B. SimPoint 3.0: Faster and More Flexible Program Phase Analysis. J Instr Level Parallelism. 2005;7.

[13] Lattner C, Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA; 2004. p. 75-88.

[14] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention is All you Need. In: Guyon I, Luxburg UV, Bengio S, Wallach H, Fergus R, Vishwanathan S, et al., editors. Advances in Neural Information Processing Systems. vol. 30. Curran Associates, Inc.; 2017. Available from: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[15] Hochreiter S, Schmidhuber J. Long Short-Term Memory. Neural Computation. 1997 11;9(8):1735-80. Available from: https://doi.org/10.1162/neco.1997.9.8.1735.

[16] Cho K, van Merrienboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, et al.. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation; 2014.

[17] Courbariaux M, Hubara I, Soudry D, El-Yaniv R, Bengio Y. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to $+1$ or -1; 2016.

[18] Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, et al. Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices. 2005;40(6):190-200.