Undergraduate Thesis

MACHINE LEARNING APPROACH TO HIGHLY BIASED BRANCH CLASSIFICATION

Christos Theodorou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2026

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Machine Learning approach to Highly Biased Branch Classification

Christos Theodorou

Supervisor

Yiannakis Sazeidis

The personal dissertation is submitted in partial fulfillment of requested obligations for receiving the degree of computer science from the department of Computer Science of the University of Cyprus

May 2026

I declare that this thesis and any accompanying software constitute my own original work, carried out in full compliance with the University of Cyprus Code of Research Ethics(https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and that I bear full responsibility for the accuracy, integrity, and validity of all content included herein.

Acknowledgements

I would like to sincerely thank my supervisor, Yiannakis Sazeidis, for his guidance, support, and invaluable knowledge throughout this project. His advice and encouragement greatly contributed to my learning and the completion of this thesis.

I am deeply grateful to my family for their constant support and for providing everything I needed throughout the years, and to my friends and colleagues for their encouragement and support during this journey.

Finally, I wish to thank Panteleimonas Chatzimiltis for his assistance with specifications and execution, which was essential to the practical success of this research.

Abstract

Computer Programs contain many conditional branches that significantly affect processor performance. Modern processors rely on branch prediction mechanisms to estimate their outcomes before executions to maintain high performance and instruction-level parallelism. Incorrect branch predictions can lead to pipeline flushes, execution stalls, and reduced overall efficiency.

A significant percentage of conditional branches have highly biased behavior meaning they tend to produce the same outcome almost all the times they executed. These branches are particularly important because can substantially improve processor performance when identifying correctly. Understanding and predicting highly biased branches can therefore contribute to reducing misprediction penalties and improving the efficiency of modern processor architectures.

This thesis investigates the use of machine learning for the analysis and classification of highly biased conditional branches. Crucially, unlike prior architectural approaches that rely on compiler-generated "static hint bits" encoded directly into the instruction format which permanently modifies the Instruction Set Architecture (ISA), this work targets a purely hardware-integrated mechanism. The goal is to utilize these static-feature classification insights to drive an autonomous predictor embedded within the CPU hardware. Dynamic execution information was collected using Intel Pin Tool instrumentation, while additional static and structural features were extracted with the help of Control Flow Graph (CFG). The model classifies branches into three categories: highly taken, highly not taken, and not biased, while also helping analyze why certain benchmark programs achieve higher prediction accuracy than others.

The evaluation was performed using benchmark applications from the SPEC CPU2017 suite. Experimental analysis was used to examine the behavior of highly biased conditional branches across different workloads and to investigate how program characteristics and extracted features affect branch classification performance.

CONTENTS

Chapter 1

Introduction

1.1 Introduction

This thesis presents a machine learning-based approach for the analysis and classification of highly biased conditional branches using static instruction-level and structural program information. The primary goal is to investigate whether program characteristics extracted from assembly-level information and Control Flow Graph (CFG) analysis can reveal highly predictable branch behavior and explain why certain branches consistently exhibit biased outcomes across different workloads. By combining execution data with structural program features, this study aims to uncover patterns in branch behavior that can inform more accurate branch prediction strategies.

Branch prediction is a critical component of modern processors, as conditional branch instructions frequently introduce control dependencies that disrupt the instruction pipeline. Incorrect predictions lead to pipeline flushes and execution stalls, which reduce instruction throughput and overall processor performance. Additionally, cold-start effects, which occur when a branch has little or no runtime history, further degrade prediction accuracy during the early stages of program execution or when new branches are encountered. Improving branch prediction and mitigating these effects is therefore essential for enhancing CPU performance.

Prior research has demonstrated the potential of static program information for improving branch prediction. Ball and Larus[4] introduced static branch prediction

techniques that analyze program structure to estimate branch behavior, while the Evidence-Based Static Branch Prediction (ESP)[5] approach uses a corpus of programs to predict branch outcomes in new applications. ESP employs machine learning methods, such as neural networks and decision trees, to map static branch features to predicted outcomes. These approaches have shown that program-level and instruction-level characteristics can provide valuable predictive information even before runtime execution occurs, complementing dynamic predictors and helping reduce mispredictions.

The methodology combines runtime branch information collected via Intel Pin Tool instrumentation with structural features extracted through CFG analysis. Instruction-level features include opcodes, execution counts, branch outcomes, and surrounding instruction context, while structural features capture control flow relationships such as dominance, loops, and successor/predecessor information. A machine learning model is then applied to classify branches into highly taken, highly not taken, and not biased categories. Feature-importance analysis further identifies which characteristics most strongly influence branch predictability, and benchmark-specific analysis explains why some programs achieve higher classification accuracy.

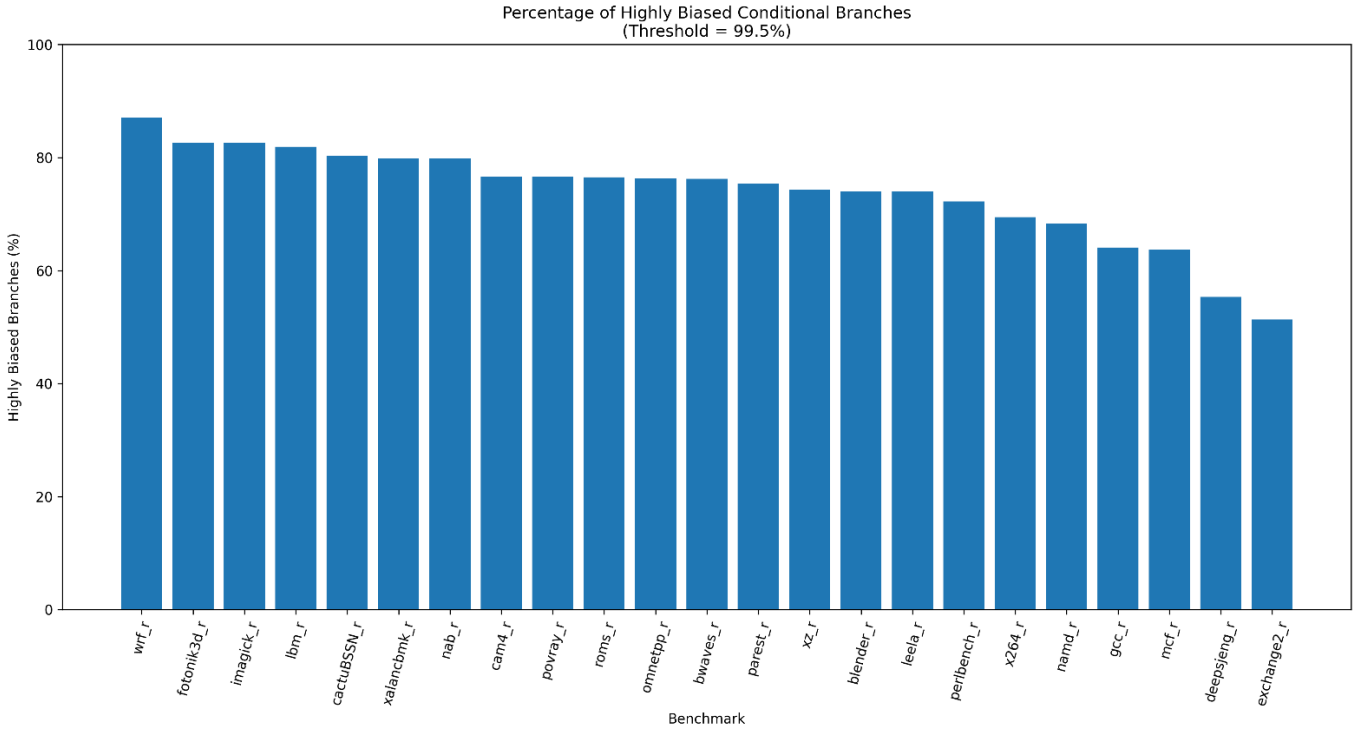


Figure 1.1: Percentage of conditional branches classified as highly biased in each SPEC CPU2017 benchmark application using a 99.5% branch bias threshold.

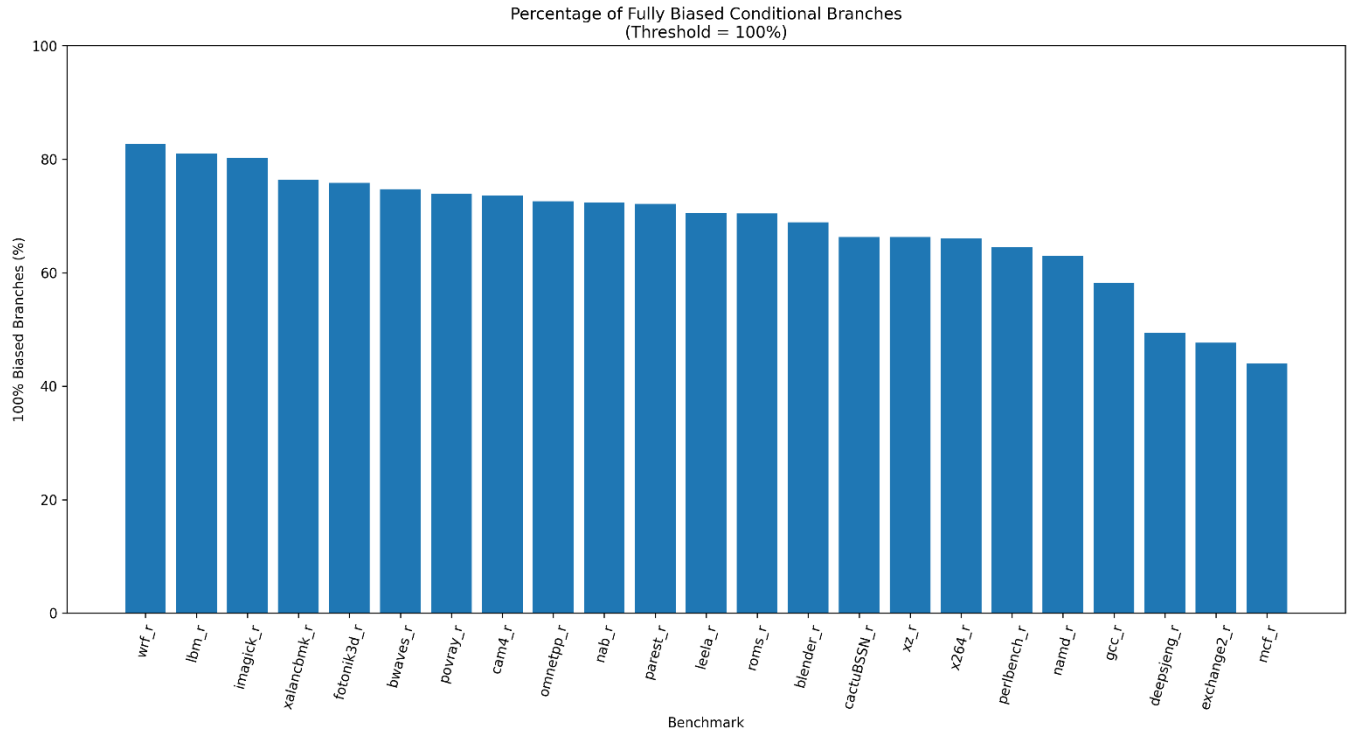


Figure 1.2: Percentage of conditional branches classified as highly biased in each SPEC CPU2017 benchmark application using a 100% branch bias threshold.

Experimental results indicate that a substantial fraction of conditional branches is highly biased across SPEC CPU2017 benchmarks. Figure 1.1 shows the percentage of branches classified as highly biased using a 99.5% threshold, while Figure 1.2 shows the stricter 100% threshold, which excludes branches exhibiting even minor variation, such as loop exit boundaries or stochastic, data-driven branching patterns. Despite the stricter criterion, many benchmarks still maintain a high proportion of fully biased branches, underscoring that a significant portion of program behavior is inherently predictable. These findings demonstrate the potential of leveraging static and structural program features to improve branch prediction, reduce pipeline stalls, and mitigate cold-start effects in modern processors.

These findings directly motivate this thesis, as they indicate that highly predictable branches can be systematically identified and analyzed using static instruction-level and structural program information. Understanding why certain branches are consistently biased and leveraging this knowledge can improve branch prediction mechanisms in modern processors, reduce pipeline stalls, and mitigate cold-start effects when limited runtime history is available. The persistence of fully biased branches across diverse workloads underscores the potential impact of this analysis on optimizing CPU performance and informing future branch prediction strategies.

Collectively, this study builds on previous static prediction research, including Ball and Larus[4] and ESP, by systematically analyzing highly biased branches and demonstrating how their predictable behavior can be exploited. The insights gained provide guidance for designing more accurate branch predictors, optimizing processor performance, and informing future hardware- and software-assisted strategies for branch prediction.

1.2 Outline

This thesis begins with Chapter 2, which provides the necessary background on processor pipelines and conditional branch prediction, including predictor designs such as bimodal, GShare, profile-based, and static predictors, as well as key challenges like aliasing, cold-start effects, and prediction limitations. Chapter 3 describes the methodology, covering benchmark applications, branch data collection with Intel Pin Tool, extraction of instruction-level branch information, and structural program analysis using CFG. Chapter 4 presents the machine learning model for classifying highly biased branches, detailing data processing, model architecture, training setup, and evaluation metrics. Chapter 5 reports the experimental results and analysis, including benchmark-specific performance, Leave-One-Benchmark-Out evaluation, and feature-importance insights. Finally, Chapter 6 concludes the thesis, summarizes the main contributions, and discusses directions for future work on improving branch prediction using static and structural program information.

Chapter 2

Background

2.1 Pipeline and Branch Prediction

When a program is executed, instructions are processed through a pipeline, a series of stages that allow multiple instructions to be in different phases of execution simultaneously. Each instruction passes through the stages of Instruction Fetch (IF), Instruction Decode/Register Read (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). By overlapping the execution of instructions, pipelining increases instruction throughput, improves hardware utilization, and allows higher clock frequencies, ultimately reducing program execution time. In a pipelined processor, the execution time of a program is approximately

$$\text{Execution Time} = (I + (n-1)) \times CT_{\text{stage}}$$

where I is the dynamic number of instructions, n is the number of pipeline stages, and CT_{stage} is the clock period determined by the longest stage.

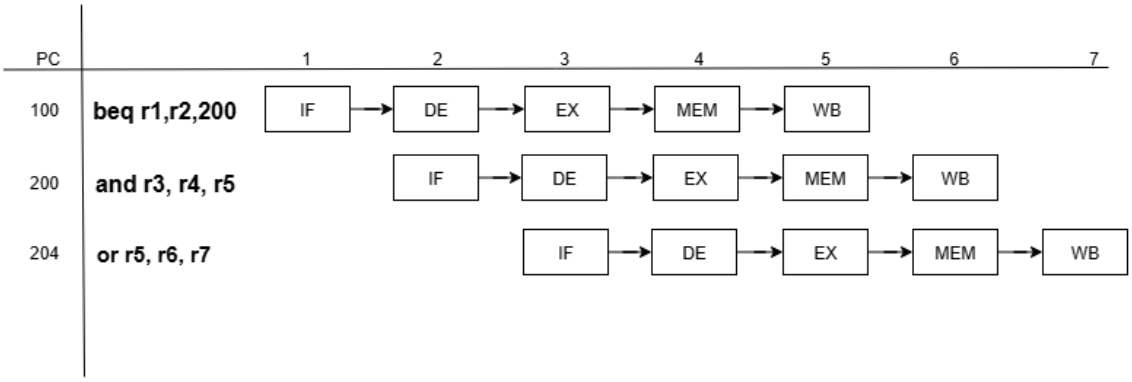


Figure 2.1: Instruction execution in a 5-stage pipeline. Each stage completes in one clock cycle (CT)

Pipelining enables multiple instructions to occupy different stages concurrently, which increases throughput without requiring each instruction to finish completely before the next instruction begins.

2.1.1 Pipeline Stages

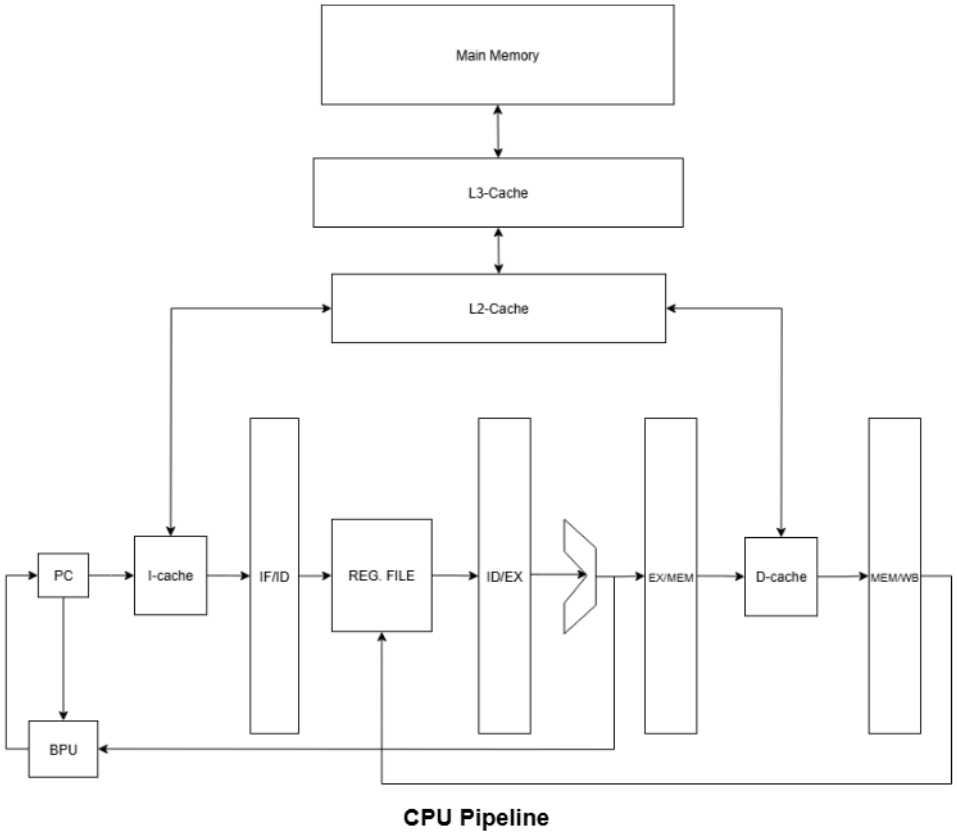


Figure 2.2: Structure of a 5-stage CPU pipeline including the memory hierarchy with L2/L3 caches and main memory

The structure in Figure 2.2 represents the stages an instruction passes through in a 5-stage CPU pipeline. In addition to the basic operation of each stage, pipelining allows multiple instructions to overlap in execution, so that while one instruction is in the Execute stage, another can be in Decode, and a third in Fetch. This concurrent execution reduces idle time for processor components and improves overall instruction throughput.

Starting with the Instruction Fetch (IF) stage, the instruction is retrieved from retrieved from I-cache or, if it's not present, from memory (L2, L3, Main memory) and the Program Counter provides the address of the instruction.

Once the instruction is fetched the Decode Stage (ID) takes part on the next clock cycle. The instruction is decoded to determine its operation type, and source operands are read from register file.

After the instruction is decoded it's ready to execute (EX) the specified arithmetic, logical or address.

If the instruction requires memory access (MEM), such as load or store, the processor reads or writes to the memory (D-Cache, L2, L3, Main Memory).
The result of the instruction is written back (WB) to the register file.

By processing multiple instructions simultaneously in different stages, the pipeline allows the processor to maximize hardware utilization and reduce the effective time per instruction. This overlapping execution forms the foundation for understanding pipeline hazards and the subsequent need for branch prediction mechanisms.

2.1.2 Control Hazards and Pipeline Stalls

Instructions that do not affect the program's control flow can progress through the pipeline simultaneously, with each instruction occupying a different stage.

However, when a conditional branch is encountered, the processor cannot immediately determine which instruction should be executed next, creating a control hazard. The next instruction depends on whether the branch is taken or not, and until the branch outcome is resolved in the executed stage, next instructions fetched might not be needed.

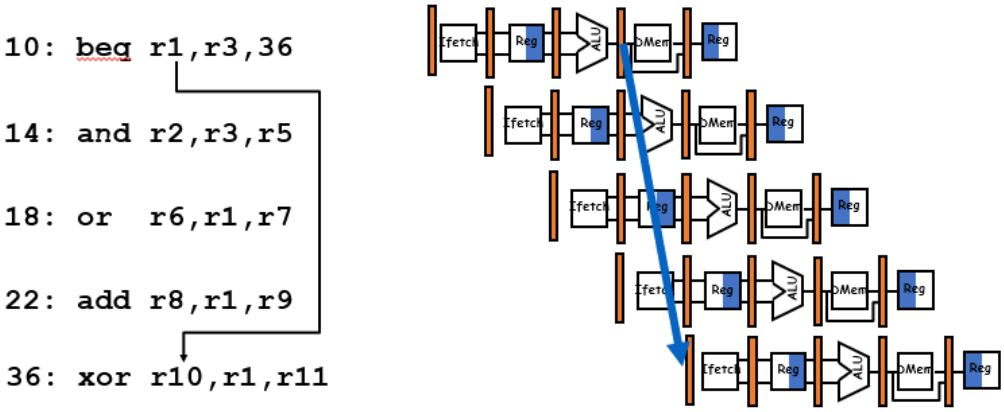


Figure 2.3: Conditional branch in a pipeline, where next instructions fetched were not needed

In figure 2.3 a conditional branch instruction is fetched and enters the pipeline like any other instruction. However, the outcome of the branch is unknown until executed, creating a control hazard. In the meantime, other instructions were fetched falsely and caused the pipeline to work without purpose on 3 instructions.

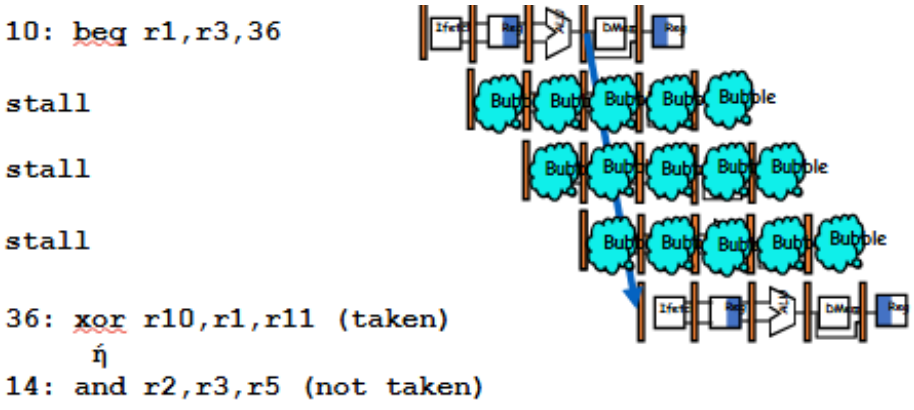


Figure 2.4: Stall technique in a pipeline when a branch is fetched

One common way to handle control hazards is the stall technique, shown in Figure 2.4. When a branch instruction is fetched, the pipeline is temporarily stalled until the branch outcome is resolved. In a typical 5-stage pipeline, this can

result in multiple cycles where no useful instruction execution occurs, reducing overall instruction throughput. The performance penalty of stalls increases with pipeline depth, as more instructions may already be in-flight and need to be flushed if the branch prediction is incorrect.

The problem with stalling technique is that always when a branch is fetched the next 3 cycles are wasted. Therefore, Modern processors often employ branch prediction mechanisms to mitigate this penalty. By speculating the likely outcome of a branch, the processor can continue fetching and executing instructions without stalling. Accurate predictions allow the pipeline to operate efficiently, while mispredictions require flushing and recovery, incurring a temporary performance cost.

2.1.3 Branch prediction Mechanism

Modern processors employ a variety of branch prediction mechanisms, including Bimodal predictors, GShare predictors, Branch Target Buffers (BTB), and Pattern History Tables (PHT/BHT), which are discussed in McFarling's seminal work (McFarling, 1993). These mechanisms allow the processor to speculate the likely outcome of a branch and continue fetching instructions along the predicted path. Correct predictions maintain pipeline efficiency, while incorrect predictions require flushing speculatively fetched instructions and restarting execution along the correct path.

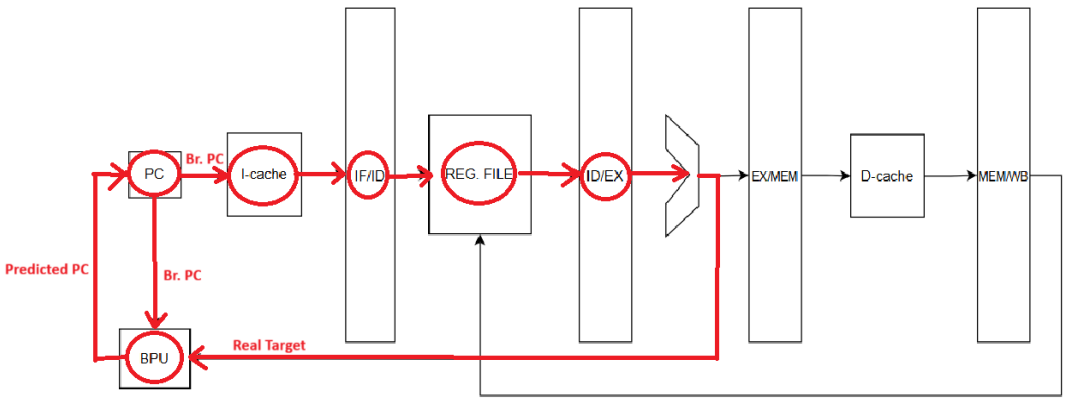


Figure 2.5: Flow of pipeline when a branch is fetched

The branch prediction unit (BPU) speculates the outcome of a conditional branch before it is resolved in the Execute stage, allowing the pipeline to continue fetching and executing instructions along the predicted path. If the prediction is correct, the pipeline proceeds without interruption, maintaining high instruction throughput. If the prediction is incorrect, the pipeline must flush speculatively fetched instructions and resume execution from the correct target, incurring a temporary penalty.

The branch prediction unit plays a central role in mitigating control hazards and maintaining pipeline efficiency. By speculating the likely outcome of conditional branches, it allows the processor to continue instruction execution without unnecessary stalls, improving throughput and reducing the impact of cold-start effects. The choice and design of branch predictors—ranging from simple Bimodal predictors to more sophisticated dynamic predictors such as GShare and profile-based mechanisms—directly influence the accuracy of speculation and the overall performance of the pipeline.

In the following section, 2.2 Branch Predictors, these different predictor types are discussed in detail, including their operating principles, strengths, and limitations, providing the foundation for understanding how highly biased branches can be leveraged to further improve prediction accuracy.

2.2 Branch Predictors

Branch predictors are hardware mechanisms designed to estimate the outcome of conditional branch instructions, allowing the processor to continue fetching and executing instructions speculatively, thereby reducing pipeline stalls and

maintaining high instruction throughput (McFarling, 1993[6], Ball & Larus, 1993,[4]Calder et al., 1997) [5]. Several branch predictors have been introduced in the literature and are commonly discussed in academic and technical works. These include Bimodal predictors, GShare predictors, Profile-Based predictors, and Static branch prediction, which rely on specialized microarchitectural structures such as branch target buffers (BTB), pattern history tables (PHT/BHT), and caches to store branch history and facilitate accurate predictions.

Ball and Larus (1993) [4] introduced static branch prediction heuristics, which use compile-time information and control-flow characteristics to predict branch behavior without relying on runtime history. These heuristics form the basis of many static prediction strategies in modern processors. Building on this idea, Calder et al. (1997) proposed evidence-based static prediction[5], which leverages machine learning techniques to analyze instruction-level and structural program features to improve prediction accuracy, particularly for branches with predictable patterns.

The predictors introduced above provide the foundation for understanding branch prediction, but they also face limitations such as cold-start effects, aliasing, and reduced accuracy for certain branch types. In this thesis, these predictors serve as a reference point to explore how static and structural program information can be used to mitigate such limitations. By analyzing and classifying highly biased branches, this work aims to improve prediction reliability, reduce stalls in the pipeline, and provide insights into why some branches are easier to predict than others, ultimately enhancing pipeline efficiency and addressing challenges that conventional predictors encounter.

2.2.1 Bimodal Branch Predictor

The bimodal branch predictor is one of the simplest forms of branch prediction used in modern processors to guess whether a branch will be taken or not taken. Its simplicity lies in maintaining a single bit or two bits of state for each branch instruction in the program, which is indexed by part of the instruction's address.

It uses a Pattern History Table (PHT) where its rows are indexed by the PC's last bits and it stores 2-bit values (00, 01, 10, 11). Those values represent the history of a branch. When a branch is taken the value is increased else the value is decreased.

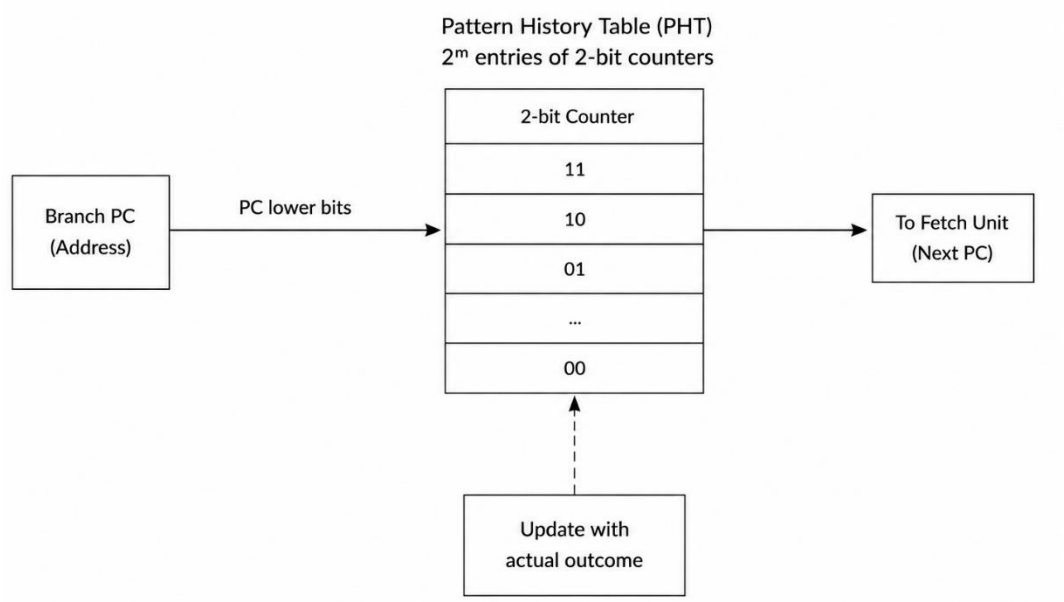


Figure 2.6: Representation of the bimodal predictor

Each value in the PHT shows the prediction, if a branch is taken or not taken. If the value stored for a branch in the PHT table is >=10 (10, 11) the prediction is taken else (00, 01) is not taken. Therefore, when a branch is fetched its indexed using the last bits with its last bits and looking at the value stored the prediction is decided. When the actual outcome of the branch is executed and available the PHT is updated.

The problem with a small table is that multiple branches may share the same counter, resulting in prediction inaccuracy.

2.2.2 GShare predictor

Another predictor is the GShare where it's an improved branch predictor compared with the Bimodal predictor. It combines global branch history with the branch's PC to index the PHT.

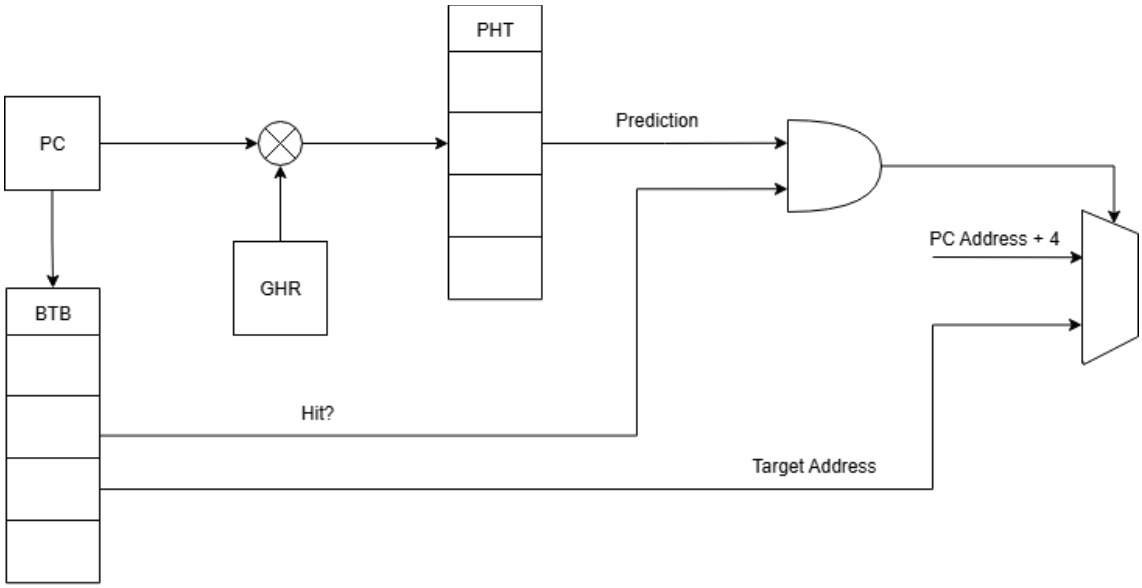


Figure 2.7: GShare branch predictor

Figure 2.7 illustrates the GShare predictor, which utilizes a Branch Target Buffer (BTB) a small cache that stores the target addresses of recently executed branch instructions. When a branch is fetched, the BTB provides a predicted target address, enabling the pipeline to speculatively fetch the next instruction. Without a valid BTB entry, the processor must stall until the branch outcome is resolved.

The Global History Register (GHR) stores the outcomes of the most recently executed branches, allowing the predictor to capture correlations between branch instructions. The branch's program counter (PC) is combined with the GHR using an XOR operation to form an index into the Pattern History Table (PHT), which predicts whether the branch will be taken or not.

When a branch is fetched, if the BTB contains a valid entry for that PC and the PHT predicts the branch will be taken, the pipeline speculatively fetches

instructions from the BTB-provided target address. Otherwise, if the BTB has no valid entry or the PHT predicts the branch will not be taken, the processor fetches the next sequential instruction. This mechanism allows GShare to minimize stalls by leveraging both historical branch behavior and target address information.

In the event of a misprediction, when the branch outcome determined in the Execute stage does not match the PHT prediction, the pipeline must flush all speculatively fetched instructions that followed the incorrect path. The program counter is then updated to the correct target address, and instruction fetching resumes from this point. Simultaneously, the PHT entry and the GHR are updated with the actual branch outcome to improve the accuracy of future predictions.

This feedback mechanism enables GShare to adapt dynamically to programming behavior, capturing both global branch correlations and local branch tendencies. By combining BTB and PHT information, GShare reduces the number of pipeline stalls and improves overall instruction throughput compared to simpler predictors such as bimodal or static branch prediction.

One of the main limitations of the Branch Target Buffer (BTB) is the cold-start problem. When a branch instruction is encountered for the first time, the BTB has no stored target address for it, and the processor cannot predict the branch outcome. As a result, instruction fetching must stall until the branch is resolved in the Execute stage, temporarily reducing pipeline throughput. This limitation is particularly significant for short-running programs or branches that are executed infrequently, where limited runtime history is available.

2.2.3 GShare vs Bimodal

The evaluation of branch predictors, as reported in *Combined Branch Predictors* (McFarling, 1993), was conducted on SPEC'89 benchmarks, running 10 million instructions per benchmark. The results indicate that prediction accuracy

improves with increasing predictor size, as larger Pattern History Tables (PHTs) help reduce aliasing.

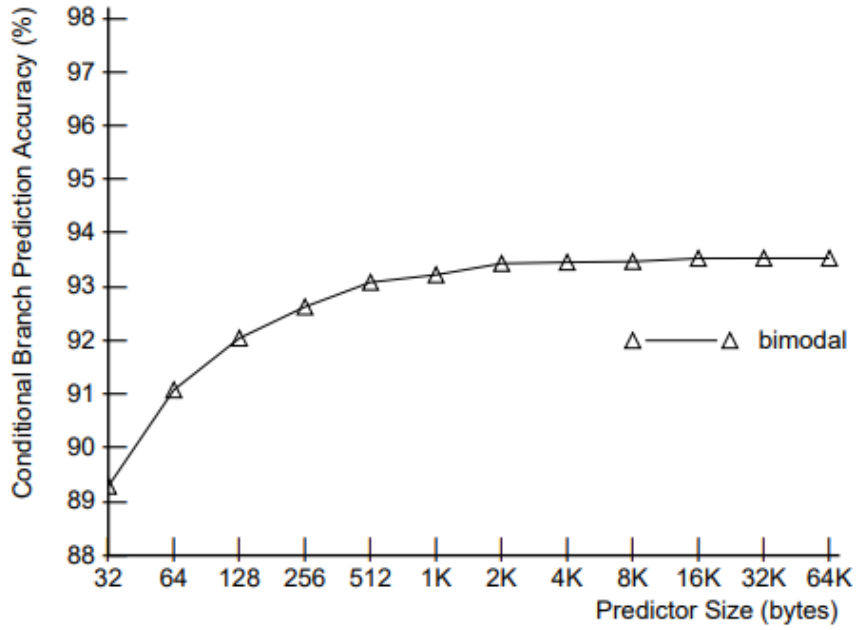


Figure 2.8: Accuracy of bimodal branch predictor as a function of PHT size[6]

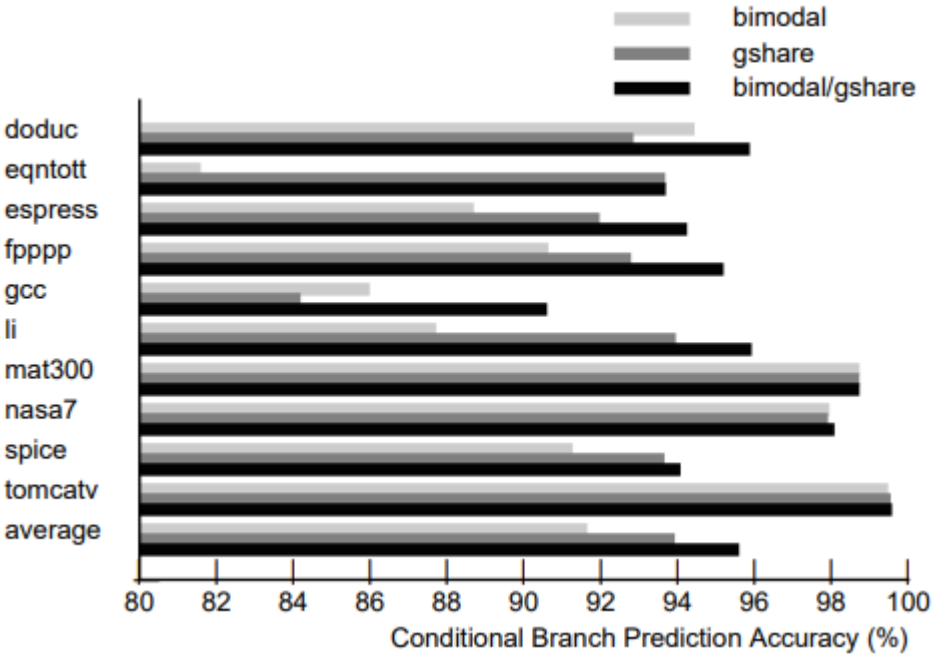


Figure 2.9: Comparison of accuracy Bimodal, Gshare[6]

Figure 2.8 illustrates how the accuracy of the Bimodal predictor increases as the PHT size grows, highlighting the importance of reducing aliasing conflicts. Figure 2.9 compares the prediction accuracy of Bimodal and GShare predictors across benchmarks, showing that GShare generally achieves higher accuracy due to its

ability to exploit global branch history.

Despite their effectiveness, both predictors face challenges with cold-start branches encountered for the first time. In the Bimodal predictor, when a new branch maps to an empty PHT entry, the default prediction is predicted as taken, leading to potential mispredictions and reduced performance. Similarly, in GShare, if a branch's address is not present in the Branch Target Buffer (BTB), the predictor defaults to not taken, resulting in lower initial accuracy for brand-new branches. These limitations emphasize the need for techniques that can mitigate cold-start effects and improve early prediction accuracy in modern processors.

2.2.4 Profile-Based predictor

Another predictor that has been proposed and widely used is the Profile-Based Branch Predictor, which relies on offline program profiling to guide predictions at runtime. While profile-based prediction can achieve very high accuracy, it requires additional steps that can be time-consuming, such as program instrumentation and sample input execution.

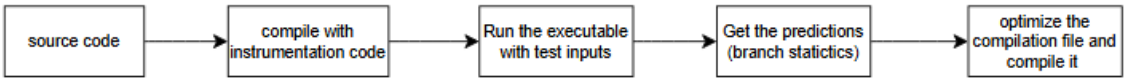


Figure 2.10: Phases of Profile-Based Branch Predictor

As illustrated in Figure 2.10, the profile-based approach begins by compiling the program with instrumentation code to monitor branch behavior. The program is then executed with representative input sets to collect statistics on the outcomes of conditional branches. Based on these observed patterns, a prediction for each branch is determined and stored for use during actual program execution.

Unlike table-based predictors, profile-based prediction does not suffer from aliasing because each branch in the executable is assigned its own prediction.

This makes it particularly effective for programs with simple or predictable control flow. However, profile-based predictors still face challenges with cold-start branches: if a branch is not executed during the profiling runs, no prediction exists for it during the actual program execution, which may reduce prediction accuracy for previously unseen paths.

2.2.5 Static Branch Prediction

In addition to dynamic and profile-based predictors, static branch prediction provides a complementary approach by leveraging compile-time heuristics. Instead of relying on runtime history or offline profiling, static predictors use information available during compilation to make predictions about branch behavior. Common heuristics include predicting backward branches as taken (typical for loops) and forward branches as not taken (typical for conditional statements).

Ball and Larus (1993) [4] introduced a set of heuristics for static prediction, demonstrating that significant prediction accuracy can be achieved by analyzing branch opcodes, loop structures, and instruction patterns at compile time. Their work also introduced evidence-based static prediction, [5] where simple machine learning models can be trained on offline execution traces to determine the most likely branch outcome for each branch type. This approach allows static predictors to incorporate observed patterns without requiring dynamic hardware tables, while remaining relatively lightweight.

A key challenge was selecting the most effective heuristics for each branch, particularly when a single branch could match multiple heuristics. Their solution was to assign a fixed order of priority to the heuristics: for each branch, the first heuristic in the predefined order that applies determines the predicted outcome.

Heuristic Name	Heuristic Description
Loop Branch	Predicting that the edge back to the loop's head is taken and that the edge exiting the loop is not taken.
Pointer	If a branch compares a pointer against null or compares two pointers, predict the branch on false condition as taken.
Opcode	If a branch checks an integer for less than zero, less than or equal to zero, or equal to a constant, predict the branch on false condition.
Guard	If a register is an operand of the branch comparison, the register is used before being defined in a successor block, and the successor block does not postdominate the branch, predict the successor block as taken.
Loop Exit	If a comparison is inside a loop, and no successor is a loop head, predict an edge exiting the loop as not taken.
Loop Header	Predict the successor that does not postdominate and is a loop header or a loop preheader as taken.
Call	Predict the successor that contains a call and does not postdominate the branch as not taken.
Store	Predict the successor that contains a store instruction and does not postdominate the branch as not taken.
Return	Predict the successor that contains a return as not taken.

Table 2.1: Heuristics used by Ball and Larus

Their Heuristic were very useful for future work and were applied somehow on a lot of predictors and they still be used. Some of them are also used in this machine learning model.

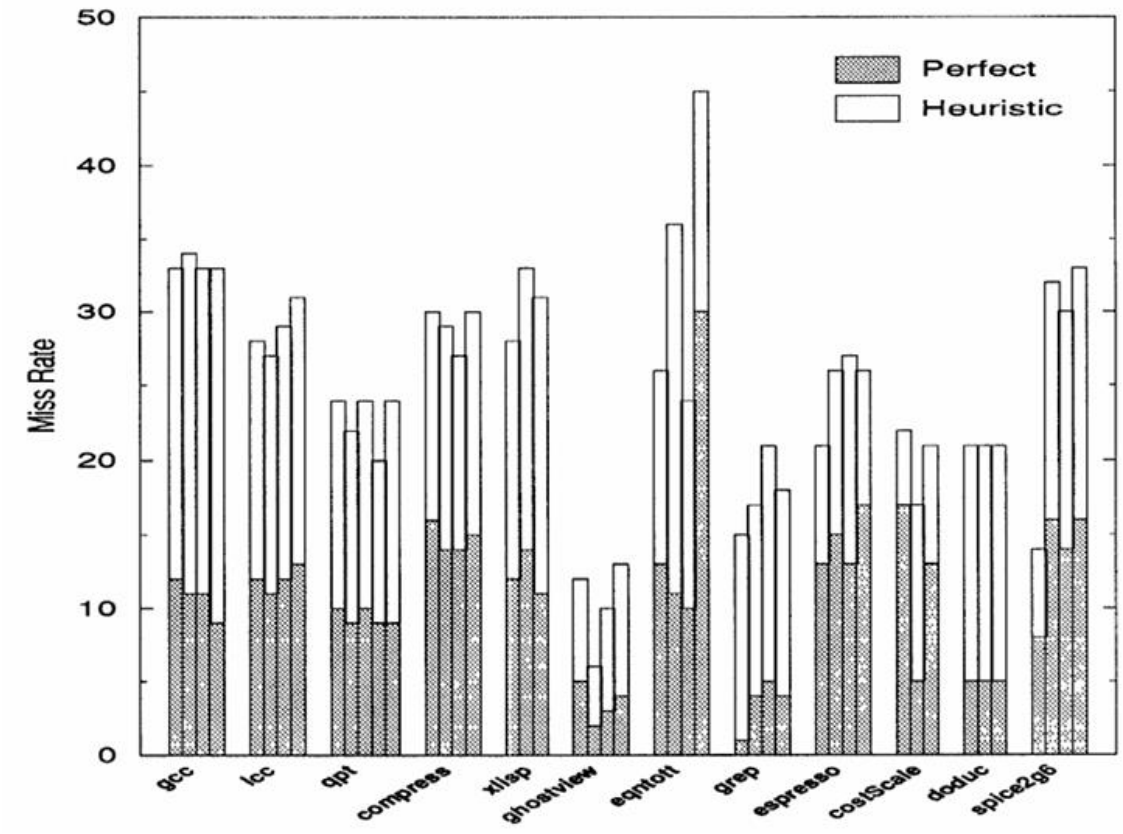


Figure 2.11: Miss rates from Ball and Larus predictor compared to the perfect prediction

Figure 2.12 shows the branch prediction miss rates for the heuristics proposed by Ball and Larus[4] compared to the "perfect" prediction scenario across different benchmark programs. The dark bars represent the miss rate if each branch were predicted perfectly, while the light bars represent the miss rates using the static heuristics.

It can be observed that although the heuristics improve branch prediction compared to random guessing, they still leave a noticeable gap from the ideal "perfect" prediction, especially in benchmarks with more complex control flow like eqntott or compress. This demonstrates the limitation of static heuristics: some branches cannot be accurately predicted using a fixed heuristic alone, and multiple heuristics may compete for the same branch. Ball and Larus [4] addressed this by prioritizing heuristics and applying only the first matching one for each branch.

Overall, the figure demonstrates that while static heuristics provide an efficient and low-overhead approach to branch prediction, they are inherently limited in accuracy compared to dynamic or profile-guided methods.

2.2.6 Evidence-Based Branch Prediction

Building on the limitations of static and profile-based predictors, researchers introduced evidence-based branch prediction, [5] which combines static program features with machine learning techniques to improve prediction accuracy. Unlike Bimodal or GShare, which rely on runtime history, or profile-based prediction, which depends on offline execution traces, evidence-based methods analyze instruction-level and structural features of branches to estimate their likely outcomes.

In this approach, their method was Collect programs and program inputs (corpus) and extract static information (static feature set). Run the program to get the dynamic behavior of each branch and store it . So now dynamic and static information is gathered and there is a relationship between them that can be used (What's the behavior of the branch and how it looks). Those data are used by

machine learning models to make predictions on branches.

Feature Number	Feature Name	Feature Description
1	Branch Instruction	The opcode of branch instruction
2	Branch Direction	F—Forward branch, B—Backward branch
3	Branch Operand Opcode	The opcode of the instruction that defines the register used in the branch instruction (or ?, if the branch operand is defined in a previous basic block)
4	Branch Operand Function	The branch function (see text)
5	Branch Operand Type	The operation type (see text)
6	RA Opcode	If the instruction in (3) uses an RA register, this is the opcode of the instruction that defines that register (? otherwise)
7	RA Function	The *RA* function (see text)
8	RA Type	The RA type (see text)
9	RB Opcode	If the instruction in (3) uses an RB register, this is the opcode of the instruction that defines that register (? otherwise)
10	RB Function	(As "RA Function" above, but for RB register)
11	RB Type	(As "RA Type" above, but for RB register)
12	Loop Header	LH—the basic block is a loop header, NLH—not a loop header
13	Language	The language of the procedure is in C or Fortran
14	Procedure Type	The branch's procedure is a Leaf, NonLeaf, or calls itself recursively (CallSelf)
	Features of the Taken Successor of the Branch	
15	Branch Dominates	D—basic block dominates this successor, or ND—does not dominate
16	Branch Postdominates	PD—the successor basic block postdominates the basic block with the branch, or NPD—does not postdominate
17	Successor Ends	Branch type ending successor basic block, possible values (FT—fall through, CBR—conditional branch, UBR—unconditional branch, BSR—branch subroutines, JUMP—jump, IJUMP—indirect jump, JSR—jump subroutines, IJSR—indirect jump subroutine, RETURN, or NOTHING)
18	Successor Loop	LH—the successor basic block is a loop header or unconditionally passes control to a basic block which is a loop header, NLH—not a loop header
19	Successor Backedge	LB—the edge getting to the successor is a loop backedge, NLB—not a loop exit backedge
20	Successor Exit	LE—the edge getting to the successor is a loop exit edge, NLE—not a loop exit edge
21	Successor UseDef	UBD—the successor basic block has a use of a register before defining it and that register was used to determine the destination of the current conditional branch instruction, NU—no use before def in successor
22	Successor Call	PC—the successor basic block contains a procedure call or unconditionally passes control to a basic block with a procedure call, NPC—no procedure call down here
	Features of the Not Taken Successor of the Branch	
23–30	Same as features 15–22 above	The same set of features (Branch Dominates through Successor Call) are collected for the not taken successor of the branch.

Table 2.2: Features used on ESP

Table 2.1 shows every feature selected for the predictor. Most of those were used in this ML approach and the features will be discussed in chapter 3.3. Their features are very useful on predicting heavily biased branches outcome, and they are a big subject on this study to find out the behavior and why some branches are heavily biased.

	Branch Prediction Miss Rates							
Program	BTFNT	APHC (B&L's)	DSHC (B&L's)	DSHC (Ours)	ESP NN	ESP DT	ESP DT (self)	Perfect
bc	40	37	35	35	32	30	27	14
bison	52	15	16	16	14	16	12	4
burg	53	35	33	32	26	22	16	9
flex	43	33	39	38	19	27	23	9
grep	42	27	23	22	19	23	15	12
gzip	33	32	33	33	20	32	11	9
indent	42	27	28	27	19	22	13	6
od	44	44	40	40	30	21	12	8
perl	35	39	36	36	26	27	13	4
sed	45	22	22	23	25	26	8	5
siod	50	34	32	33	27	30	15	10
sort	44	35	41	42	21	19	18	8
tex	43	37	38	36	30	31	19	13
wdiff	42	32	11	11	4	11	8	3
yacr	32	14	11	12	14	13	8	6
Other C Avg	43	31	29	29	22	23	15	8
alvinn	2	2	2	2	1	2	2	0
compress	44	25	26	28	30	28	14	14
ear	10	8	8	8	8	8	7	7
eqntott	47	7	7	7	6	5	3	2
espresso	34	24	23	23	32	25	18	15
gcc	48	34	35	34	31	32	27	12
li	43	26	25	27	28	29	13	12
sc	39	29	31	29	24	26	16	9
SPEC C Avg	34	19	20	20	20	19	12	9
doduc	23	19	20	19	16	39	10	5
fpppp	42	53	52	52	35	22	14	11
hydro2d	28	17	16	16	12	10	5	4
mdljsp2	69	41	62	62	64	65	10	10
nasa7	8	12	12	11	5	5	5	3
ora	46	18	18	18	18	12	5	5
spice	16	16	18	14	14	11	11	7
su2cor	17	21	20	20	12	12	11	10
swm256	1	1	1	1	1	1	1	1
tomcatv	44	44	44	44	1	44	1	1
wave5	19	27	24	23	21	24	9	6
SPEC Fortran Avg	29	24	26	25	18	22	7	6
APS	28	30	34	31	26	29	17	10
CSS	39	29	40	36	33	25	24	9
LWS	38	32	25	25	18	17	17	16
NAS	42	12	22	22	12	12	6	4
OCS	4	6	5	5	4	6	4	2
SDS	18	32	25	19	21	22	15	12
TFS	12	10	15	13	11	11	8	6
TIS	18	26	25	22	16	16	16	16
WSS	32	28	26	26	25	24	25	11
Perf Club Avg	26	23	24	22	18	18	15	10
Overall Avg	34	25	26	25	20	21	12	8

Table 2.3: Branch predictors miss rates on SPECS

In the table above, branch prediction miss rates are presented across a wide range of benchmarks for several branch predictors, including Bimodal, GShare, and the Evidence-Based Static Predictor (ESP) [5]. In this study, the ESP miss rates were obtained using a leave-one-out evaluation method, in which all benchmarks except one are included in the training dataset, and the remaining benchmark is used for testing. Three variations of ESP are reported: ESP Neural Network (NN), ESP Decision Tree (DT), and ESP DT (self), all evaluated on the same test dataset. Overall, ESP NN achieved slightly better performance than the DT variants, with a miss rate of 20% compared to 21% for ESP DT.

The significance of ESP lies not only in its competitive accuracy compared to other predictors but also in its ability to substantially mitigate the cold-start problem. Unlike traditional predictors, ESP does not rely on fixed rules, instead it represents each branch using a rich set of features, allowing for more flexible and informed predictions.

2.3 Challenges – Limitations

Through decades of research and continuous improvements in branch prediction, many problems have been addressed, while others remain unresolved. Several challenges continue to limit predictor accuracy—even modern approaches that leverage machine learning, the most powerful tool of today's technology, cannot achieve perfect predictions. Nevertheless, the pursuit of better understanding and optimization drives ongoing research in this field. This thesis, therefore, focuses specifically on highly biased branches, aiming to analyze and understand their behavior to improve prediction accuracy and mitigate performance penalties in modern processors.

2.3.1 Aliasing

Aliasing was one of the problems that was faced by Bimodal predictor. It's the issue where multiple branches can have the same index and share the same counter in PHT or BTB table. Predictors such as GShare reduced the aliasing by combining GHR with PC when computing table indices. Gshare reduced but did not eliminate this problem. Profile-Based and Program-Based predictions do not have the aliasing problem, because for every branch of an executable a prediction is decided.

2.3.2 Cold-Start Effects

Cold-start effects in branch prediction occur when a processor encounters branch instructions for which no prior execution history or profiling data exists. This situation is particularly evident at the beginning of program execution or when a core transitions from a low-power state (C-state), where the branch predictor and instruction caches are initially empty. As shown in Figure 2.12, these cold-start conditions can introduce significant delays, measured using cycles per instruction (CPI) and misses per kilo-instruction (MPKI) across a variety of benchmarks.

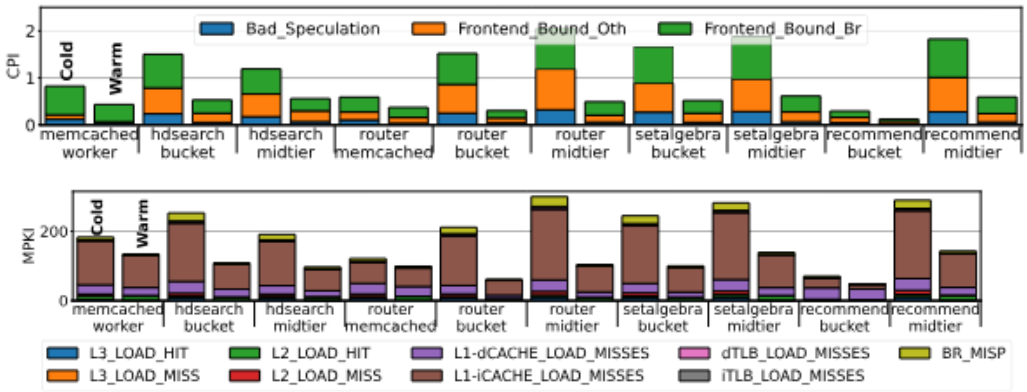


Figure 2.12: CPI and MPKI for Branch prediction and Frontend Stalls

Figures 2.12, adapted from the work of G. Antoniou et al. (2023),[3] illustrate the impact of cold-start branches on frontend performance. In the CPI breakdown, the Frontend_Bound_Br component, representing cycles consumed by unresolved branch instructions, dominates during cold-start conditions, indicating that branch predictors initially lack sufficient information, causing the pipeline to stall. The MPKI analysis further highlights that branch mispredictions are a

substantial source of frontend latency. Specifically, cold-start conditions lead to higher misprediction rates, which can trigger instruction flushes and exacerbate delays across instruction caches (L1I, L1D) and translation lookaside buffers (TLBs). Across benchmarks, the performance penalty from cold-start branch mispredictions ranges from 46% to 233%, with an average impact of approximately 118% relative to warm execution.

These results demonstrate that branch prediction is a critical contributor to performance degradation in cold-start scenarios. The figures highlight that traditional predictor, when facing previously unseen branches, incur significant overheads before sufficient history is accumulated. This motivates the use of advanced approaches, such as evidence-based predictors and machine-learning models, which can leverage static and dynamic program features to provide initial branch predictions, thereby mitigating the cold-start penalty. By addressing this challenge, processors can maintain higher instruction throughput and reduce pipeline stalls during early execution or post-C-state recovery, which is especially important for latency-sensitive workloads and microservices environments.

2.3.3 Prediction Accuracy Limitations

Branch prediction accuracy is inherently limited by a combination of architectural, programmatic, and workload-related factors. Even modern predictors, including machine learning-based approaches, cannot achieve perfect accuracy due to the dynamic and often unpredictable behavior of conditional branches. Table 2.1 summarizes the main limitations for several common predictors, including Bimodal, GShare, Profile-Based, and Program-Based (Evidence-Based) approaches.

Branch Predictor	Aliasing	Cold start
Bimodal	Yes	Yes
GShare	Reduced but yes	Yes
Profile-Based	No	Yes
Program-Based	No	Reduced a lot but yes

Chapter 3

Methodology

3.1 Introduction

The goal of this study is to analyze and classify highly biased conditional branches in modern programs. A robust feature set is required for training a machine learning model capable of accurately predicting branch behavior. To this end, a combination of dynamic execution data and structural program information was collected and analyzed.

Dynamic information was obtained using Intel Pin Tool version 3.31[7], which instruments programs at the x86 instruction set architecture (ISA) level. Pin records detailed information about each conditional branch, including its program counter, branch outcome (taken or not taken), and relevant contextual instruction features. This instrumentation provides a comprehensive view of runtime branch behavior, which is critical for building predictive models.

To complement dynamic features, structural program information was also considered. Initially, a Dynamic Control Flow Graph (DCFG) analysis was attempted to extract execution paths of the programs. However, this approach was abandoned because DCFGs only capture paths that are executed during a run, leaving out unexecuted paths. This incompleteness could leak information to the model and artificially improve prediction performance. As a result, the study shifted to static Control Flow Graph (CFG) analysis using the angr[2] framework

in Python. The CFG captures all possible execution paths, including those not taken during execution, and provides key structural features such as loop headers, backedges, and basic block connectivity. By combining dynamic and static features, the model receives a rich representation of branch behavior across the program.

The dataset for this study consists of benchmark programs from the SPEC CPU2017 suite, chosen to provide diverse workloads and a wide range of branch behavior patterns. All benchmarks were executed under control conditions to collect runtime traces, while static CFGs were built from the same binaries to extract structural features. This combination ensures that the resulting dataset covers both the actual execution patterns of branches and their potential structural contexts, forming a solid foundation for training and evaluating machine learning models on highly biased branches.

To ensure the reproducibility of the data collection framework, the custom Intel Pin instrumentation tool and the *angr*-based static analysis scripts have been made publicly available. The open-source repository provides the complete source code: *https://github.com/christheo03/BranchInformations.git*

3.2 Benchmark applications

Data was collected from 22 C, C++, and Fortran programs from the SPEC CPU2017 benchmark suite. All programs were compiled using GCC version 4.8.5 in static mode for a single-core execution environment, ensuring that the binaries contained all required libraries and could be executed without dynamic dependencies. The static compilation and single-core configuration allowed for consistent and reproducible measurements across benchmarks.

Benchmark	Type	Application Area and Description
gcc_r	Integer	GNU C compiler. Compiles C programs.
perlbench_r	Integer	Perl interpreter. Executes Perl scripts.
mcf_r	Integer	Route planning (orienteering); finds shortest paths in a road network.
povray_r	Integer	Ray tracing; generates 3D graphics using C and C++.
omnetpp_r	Integer	Discrete event network simulation (C++).
bwaves_r	Floating Point	Explosion modeling using Fortran.
cactuBSSN_r	Floating Point	Physics simulation; solves Einstein's equations (C++, C, Fortran).
namd_r	Floating Point	Molecular dynamics simulation (C++).
parest_r	Floating Point	Biomedical imaging: optical tomography using finite elements (C++).
lbm_r	Floating Point	Fluid dynamics using the Lattice Boltzmann method (C).
wrf_r	Floating Point	Weather forecasting simulation (Fortran, C).
xalancbmk_r	Integer	XML to HTML conversion using XSLT (C++).
x264_r	Integer	Video compression using H.264/AVC codec (C).
blender_r	Floating Point	3D rendering and animation (C++, C).
cam4_r	Floating Point	Atmosphere modeling (Fortran, C).
deepsjeng_r	Integer	AI: alpha-beta tree search (Chess) (C++).
imagick_r	Floating Point	Image manipulation (C).
leela_r	Integer	AI: Monte Carlo tree search (Go) (C++).
nab_r	Floating Point	Molecular dynamics simulation (C).
exchange2_r	Floating Point	Recursive solution generator (Sudoku) (Fortran).
fotonik3d_r	Floating Point	Computational electromagnetics (Fortran).
roms_r	Floating Point	Regional ocean modeling (Fortran).
xz_r	Integer	General data compression using LZMA (C).

Table 3.1: SPEC2017 applications that were used in the study

Name	Instructions Executed	Executed Cond. Br.	Unique Cond. Br. Executed	Biased Cond. Br. (thr = 99.5%)	Non Biased Cond. Br.
gcc_r	198 billion	34.5 billion (17%)	86513	55467 (64.11%)	31048 (35.89%)
perlbench_r	1.242 trillion	174 billion (14%)	17137	12378 (72.23%)	4754 (27.75%)
mcf_r	27 billion	5 billion (19%)	309	197 (63.75%)	112 (36.25%)
povray_r	3.5 trillion	334.7 billion (9%)	7297	5589 (76.59%)	1708 (23.41%)
omnetpp_r	1.08 trillion	146.7 billion (13%)	8930	6813 (76.29%)	2129 (23.82%)
bwaves_r	1.4 trillion	156 billion (10%)	2274	1734 (76.25%)	540 (23.75%)
cactuBSSN_r	1.5 trillion	19 billion (1%)	11586	9305 (80.31%)	2291 (19.75%)
namd_r	2.6 trillion	38.4 billion (1%)	3505	2396 (68.38%)	1117 (31.82%)
parest_r	3.6 trillion	458.1 billion (12%)	15557	11725 (75.37%)	3833 (24.63%)
lbm_r	1.6 trillion	11.6 billion (0.7%)	1023	838 (81.92%)	182 (17.95%)
wrf_r	3.7 trillion	302.1 billion (8.16%)	40241	35049 (87.10%)	5193 (12.91%)
xalancbmk_r	1.3 trillion	346 billion (26.3%)	13595	10861 (79.89%)	2734 (20.11%)
x264_r	320.4 billion	11.7 billion (3.65%)	4480	3111 (69.44%)	1369 (30.56%)
blender_r	1.8 trillion	257 billion (14.28%)	14864	10999 (74.00%)	3865 (26.00%)
cam4_r	2.8 trillion	289.1 billion (10.33%)	23068	17678 (76.63%)	5390 (23.37%)
deepsjeng_r	1.9 trillion	186.6 billion (9.82%)	2629	1455 (55.34%)	1174 (44.66%)
imagick_r	4.4 trillion	383.3 billion (8.71%)	4052	3349 (82.65%)	699 (17.25%)
leela_r	2.2 trillion	237.1 billion (10.78%)	4077	3017 (74.00%)	1060 (26.00%)
nab_r	2.1 trillion	201.3 billion (9.59%)	2861	2365 (79.87%)	596 (20.13%)
exchange2_r	6.6 trillion	638.9 billion (9.68%)	3267	1677 (51.33%)	1588 (48.49%)
fotonik3d_r	2.2 trillion	42.0 billion (1.91%)	4752	3929 (82.68%)	823 (17.32%)
roms_r	2.9 trillion	189.3 billion (6.53%)	6838	5231 (76.50%)	1607 (23.50%)
xz_r	397.2 billion	43.9 billion (11.05%)	2128	1581 (74.30%)	547 (25.70%)

Table 3.2: Statistics about Conditional Branches Executed

For every benchmark application, detailed statistics were collected to characterize branch behavior. Specifically, Instructions Executed and Executed Conditional Branches were measured using perf, a Linux performance monitoring tool that provides hardware-level event counting with minimal runtime overhead. Perf allows precise measurement of runtime statistics, including instruction counts, branch executions, cache accesses, and other hardware events, without requiring modifications to the program source code.

To measure conditional branches, the following perf command was used:

taskset -c 0 perf stat -r 10 -e br_inst_retired.cond -- <benchmark_executable>

This command binds the benchmark to a single CPU core with taskset -c 0 and runs it 10 times, counting all retired conditional branch instructions using br_inst_retired.cond

Similarly, the total number of instructions executed was obtained with:

taskset -c 0 perf stat -r 10 -e instructions -- <benchmark_executable>.

Validation of the instrumentation framework was performed by comparing the conditional branch counts extracted by the PIN tool against hardware performance counters via perf. Taking the povray_r benchmark as a representative example, perf measured 334 billion executed conditional branches, whereas the PIN tool exhibited a minor deviation of approximately (0.03%). This marginal variation is statistically insignificant and expected, while perf relies on hardware-level architectural counters that can occasionally capture kernel-space events or speculative execution path artifacts, PIN operates via user-space dynamic binary translation. The close alignment between the two tools confirms the reliability of the PIN-based instrumentation.[7]

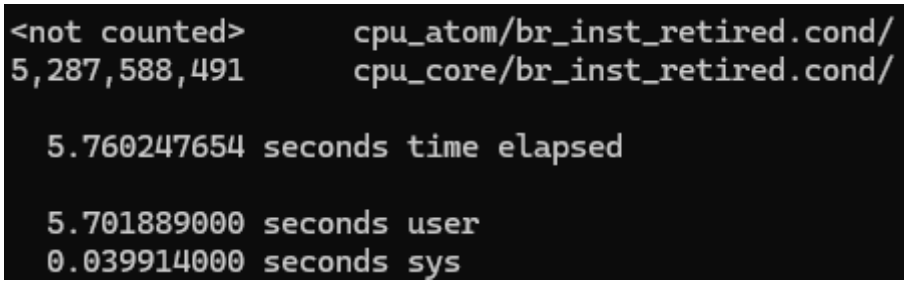


Figure 3.1: Conditional Branches executed, measured by perf

Figure 3.2: Conditional Branches executed, measured by PIN

The number of unique conditional branches executed was computed using the PIN tool. For every unique conditional branch instruction encountered at runtime, PIN recorded its total execution count alongside the number of times it resolved as taken. Using these raw counts, a directional bias rate was calculated. If a branch's rate of being taken was higher than 0.995, or conversely, lower than 0.005 (meaning it was not-taken more than 99.5% of the time), it met the criteria for the highly biased category. Static branches satisfying this condition were classified under the biased column, while the remaining branches were categorized as non-biased. [7]

An analysis of the table data shows big differences in how these benchmarks behave. Total instructions range from just 27 billion for mcf_r to 6.6 trillion for exchange2_r. Some programs are very heavy on decision-making, like xalancbmk_r where conditional branches make up 26.3% of all instructions, while others like lbm_r barely use them at 0.7%. We also see a major split in the number of unique conditional branches. Large programs like gcc_r and wrf_r use a massive number of unique branches (86,513 and 40,241), which easily overload the processor's BTB. In contrast, mcf_r reuses a tiny set of just 309 unique branches repeatedly. Finally, using the 99.5% bias rule shows that some workloads are highly predictable. For example, 87.10% of the branches in wrf_r almost always go the same way. However, programs like exchange2_r and deepsjeng_r are much harder to predict because nearly half of their branches (48.49% and 44.66%) constantly switch direction, creating a tough test for hardware branch predictors.

3.3 PIN Instrumentation Framework

To analyze fine-grained control flow and branch predictability behavior beyond simple global hardware events, a custom tool was developed using the Intel PIN dynamic binary instrumentation framework. PIN allows for the injection of arbitrary analysis code into a compiled binary at runtime via dynamic binary translation. The main entry point of the developed Pin tool hooks into image load events *(IMG_AddInstrumentFunction)* and basic block traces *(TRACE_AddInstrumentFunction)*, mapping program logic directly as instructions are translated. [7]

The core data structures rely on an efficient indexing system using the instruction program counter (PC) address as a key. For tracking, an *unordered_map<ADDRINT, branch> br_info* structure is allocated to store the specific metadata of every unique conditional branch. As execution finishes, the Fini callback function processes this map and flushes the accumulated metrics directly into a structured comma-separated values (*branches.csv*) report for post-processing evaluation.

3.3.1 Extraction of Static Branch Footprints

The extraction of the static conditional branch footprint is performed during the main instruction evaluation loop inside the *ImageLoad* routine. As the main executable image loads into memory, the tool iterates sequentially through sections, functions, and their underlying instruction streams. The tools filter for conditional control flow pathways by checking if an instruction belongs to the specific conditional branch family using the *XED_CATEGORY_COND_BR* attribute filter. Each time a unique conditional branch is discovered, a new static tracking is instantiated within the *br_info* table. The total count of distinct entries populated inside this map corresponds directly to the unique conditional branches executed metric.

Beyond simply counting unique branches, the tool extracts a comprehensive set of architectural features at each branch site to build a rich structural and contextual profile of the program. When a conditional branch is encountered, the tool records its primary location data, surrounding instructions, flag dependencies, and register-level data flows. A complete breakdown of these twenty-two extracted data fields is organized and detailed in Table 3.3.

Feature Name	Description
PC Address	The unique program counter (hexadecimal memory address) of the static conditional branch instruction.
Br Opcode	The short string representation of the conditional branch's machine opcode (e.g., `JZ`, `JNZ`, `JG`).
Executed	The total dynamic counter tracking how many times this specific branch instruction was reached during runtime.
Taken	The dynamic counter tracking the total number of times the branch condition evaluated as true and took the target path.
Offset	The relative byte distance between the current branch PC address and its target destination address.
Size	The machine instruction length measured in bytes.
Set_flag_instr PC	The program counter address of the preceding instruction that writes to the `GFLAGS` register used by this branch.
Set_flag_instr Opcode	The machine opcode of the instruction responsible for setting the status flags.
Same BBL	A boolean value indicating whether the flag-setting instruction and the branch reside within the exact same Basic Block.
Routine_type	Classifies the containing function execution structure as either a *Leaf*, *Non-Leaf*, or *Recursive* routine.
Regs_read	A list of the source CPU architectural registers read by the flag-modifying instruction, paired with the specific machine opcode that defined each register within the local basic block (or tagged as `-1` if defined externally).
Regs_write	A list of the destination CPU architectural registers written to by the flag-modifying instruction.
Prev_Instr Op (1–5)	The short string opcodes and sizes of the five instructions immediately preceding the conditional branch in static memory.
Next_Instr Op (1–5)	The short string opcodes and sizes of the five instructions immediately following the conditional branch in static memory.

Table 3.3: Features extracted from PIN tool

The extraction pipeline begins the moment a conditional branch instruction is isolated. The tool immediately extracts its inherent physical properties directly from its binary encoding block to establish the baseline identity of the branch. This step captures its unique memory PC Address using *INS_Address(ins)*, which returns the hexadecimal runtime address of the instruction. The structural mnemonic type is recorded as the Br Opcode via *OPCODE_StringShort(INS_Opcode(ins))*, providing a clean string representation of the specific branch condition (e.g., *JZ, JG*). To maintain layout awareness, the instruction Size in bytes is retrieved natively using *INS_Size(ins)*. Finally, if the branch controls a direct control flow pathway, its relative spatial target Offset is calculated mathematically by subtracting the branch's own address from its target destination using the relation:

Offset = INS_DirectControlFlowTargetAddress(ins) -INS_Address(ins)

Once this base entry is stabilized, the tool broadens its scope to parse the surrounding local neighborhood context. It looks backward in static memory to catalog the context window of five preceding instructions (*Prev_Instr Op*), utilizing *INS_Prev(ins)* to step sequentially through previous instructions and retrieving their string mnemonics via *INS_Opcode*. In an identical manner, the pipeline sweeps forward in static memory to catalog the context window of five succeeding instructions (*Next_Instr Op*), leveraging *INS_Next(ins)* to advance the instruction pointer and fetching the trailing operational behaviors via *INS_Opcode*.

Next, the extraction path shifts inward to track control flag states and structural function blocks. Because conditional branches rely entirely on flag status updates to determine if they should fire, the tool looks backward to identify the closest preceding instruction that explicitly targets the CPU's flags register. It records this instruction's identity by continually caching a pointer whenever *INS_RegWContain(ins, REG_GFLAGS)* evaluates to true, capturing the *Flag_Write_PC* via *INS_Address(flag_write)* and the *Flag_Instr_Opcode* via *OPCODE_StringShort(INS_Opcode(flag_write))*. Concurrently, the tool determines basic block locality using a trace-level instrumentation hook. By capturing every basic block (BBL) via *TRACE_BblHead* and *BBL_Next*, the tool maps each

instruction address to a unique tracking index (bbl_id). At application teardown, a boolean equivalence check computes *Same_BBL* by checking if the block ID of the branch matches the block ID of the flag-setting instruction, highlighting whether this flag modification happens safely within the branch's own straight-line execution block.

Simultaneously, the pipeline scales upward to scan the broader containing function envelope to evaluate the function's overall structure, assigning a *Routine_type* designation as described in *Algorithm 1*. The tool audits the active routine container (*RTN rtn*) during the image load phase by checking for the presence of call instructions using *INS_IsCall(ins)*. If a function contains no call instructions whatsoever, it is classified as a *Leaf* routine. If it contains call sequences, the tool actively checks if any are direct control flow paths targeting the function's own starting entry pointer using *INS_DirectControlFlowTargetAddress(ins) == RTN_Address(rtn)*. If a self-targeting loop is detected, it is labeled *Recursive*; otherwise, the routine type is recorded as *NonLeaf*.

Algorithm 1: Routine_type Attribute Computation

Input: Routine container *rtn*
Output: Classification string for *b.routine_type*

```
hasCall ← false, isRecursive ← false
rtn_entry ← RTN_Address(rtn)
foreach Instruction ins ∈ rtn do
    if INS_IsCall(ins) then
        hasCall ← true
        if INS_IsDirectControlFlow(ins) and
          INS_DirectControlFlowTargetAddress(ins) == rtn_entry then
            isRecursive ← true
        end
    end
end
if isRecursive then
    return "Recursive"
end
else if hasCall then
    return "NonLeaf"
end
else
    return "Leaf"
end
```

Deepening this data-flow audit, the tool captures the inputs (*Regs_Read*) and updates (*Regs_Write*) of the flag-setting instruction by looping through its read and written registers using *INS_RegR* and *INS_RegW*. These are converted to text strings via *REG_StringShort*.

To trace register ancestry, global tracking dictionaries (*last_writer_pc* and *last_writer_op*) are updated using *REG_FullRegName* whenever any instruction is written to a register. When the flag-writing instruction reads a register, these maps are queried to extract the *Regs_read Def Opcode*. If the register's defining instruction and the branch share the same basic block ID, its short opcode name is stored using *OPCODE_StringShort*; otherwise, it is flagged as -1 to denote an out-of-block dependency.

3.3.2 PIN Tool Data Example

To demonstrate and validate the feature extraction accuracy of the instrumentation pipeline, a test case was analyzed using a standalone function named check_value. The program was compiled into a static, non-PIE binary configuration and executed with two separate input distributions: (-5, 5) and (100, 12).

The disassembled layout of the compiled target routine is presented below:

```
0000000000401126 <check_value>:
  401126:       sub    $0x8,%rsp
  40112a:       cmp    $0x9,%esi
  40112d:       ja     40114a <check_value+0x24>
  40112f:       test   %edi,%edi
  401131:       js     401156 <check_value+0x30>
  401133:       cmp    $0xcccccccc,%edi
  401139:       jg     401162 <check_value+0x3c>
  40113b:       mov    $0x402050,%edi
  401140:       callq  401030 <puts@plt>
  401145:       add    $0x8,%rsp
  401149:       retq
  40114a:       mov    $0x402010,%edi
  40114f:       callq  401030 <puts@plt>
  401154:       jmp    401145 <check_value+0x1f>
  401156:       mov    $0x402023,%edi
```

```
40115b:      callq  401030 <puts@plt>
401160:      jmp    401145 <check_value+0x1f>
401162:      mov    $0x402035,%edi
401167:      callq  401030 <puts@plt>
40116c:      jmp    401145 <check_value+0x1f>
```

The resulting execution log generated by the custom Intel PIN tool tracking pipeline captures exactly two active conditional records, as detailed in Table 1 and Table 2.

Address	Opcode	Exec.	Taken	Offset	Size	Flag Write PC	Flag Op	Same BBL	Routine	Regs Read	Regs Write
0x40112d	JNBE	2	1	29	2	0x40112a	CMP	1	NonLeaf	esi(-1)	rflags
0x401131	JS	1	1	37	2	0x40112f	TEST	1	NonLeaf	edi(-1) edi(-1)	rflags

Address	**Prev_Op_1 (Sz)**	**Prev_Op_2 (Sz)**	**Prev_Op_3 (Sz)**	**Prev_Op_4 (Sz)**	**Prev_Op_5 (Sz)**
0x40112d	CMP (3)	SUB (4)	INVALID (0)	INVALID (0)	INVALID (0)
0x401131	TEST (2)	JNBE (2)	CMP (3)	SUB (4)	INVALID (0)
Address	**Next_Op_1 (Sz)**	**Next_Op_2 (Sz)**	**Next_Op_3 (Sz)**	**Next_Op_4 (Sz)**	**Next_Op_5 (Sz)**
0x40112d	TEST (2)	JS (2)	CMP (6)	JNLE (2)	MOV (5)
0x401131	CMP (6)	JNLE (2)	MOV (5)	CALL_NEAR (5)	ADD (4)

3.4 Structural Program Analysis

Accurate branch prediction relies not only on historical execution data but also on structural information about the program. Understanding the program's control flow allows a predictor to reason about potential execution paths even before they are dynamically observed. Structural features provide context about branch locations, their relationships with other instructions, and possible execution paths, which is particularly useful when predicting highly biased or infrequently executed branches.

In this study, the Angr[2] framework was used to create a Control Flow Graph (CFG) of the executables to collect the structural data. Angr[2] allows the analysis of binary executables, extracting nodes (basic blocks) and edges (control transfers) that form the CFG, which is essential for modeling the relationships between branches and their surrounding instructions.

A Control Flow Graph (CFG) is a directed graph where nodes represent basic blocks and edges denote possible control transfers between blocks. An edge from block *A* to block *B* indicates that *B* may execute immediately after *A*. CFGs provide a structural representation of the program, capturing all potential execution paths statically, independent of a particular input or dynamic execution. This structural information is crucial for branch prediction, as it allows predictors to infer branch behavior not only from past executions but also from program structure, improving the accuracy for highly biased or rarely executed branches.

A basic block is defined as a straight-line sequence of instructions with a single-entry point and a single exit point, such that the control flow enters at the first instruction and exits at the last without branching except at the end (Aho et al., 2007). Each basic block guarantees that all instructions inside it are executed sequentially whenever the block is entered. This property makes basic blocks the fundamental unit for constructing a program's control flow analysis.

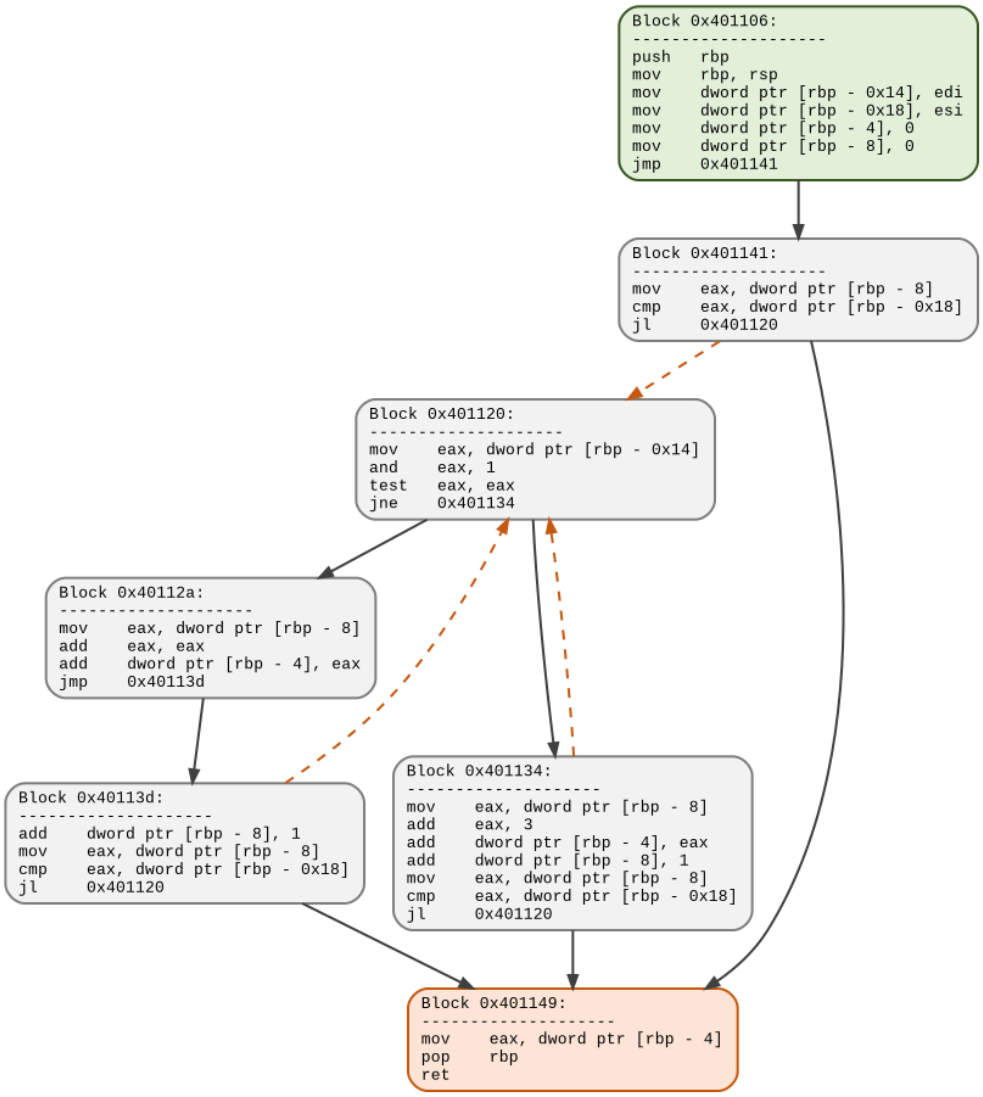


Figure 3.3: Control Flow Graph generated using angr

Figure 3.3 shows the execution flow inside a single function, where each rounded rectangle is a Basic Block containing a straight sequence of assembly instructions that ends with a Control Flow Instruction (like jmp, jl, or ret) to determine the next step. The Green block is the single entry point, the Gray blocks represents the basic blocks inside the function, and the Orange block is the final exit point where all paths merge to return the result. The Solid Black Arrows show the taken and fallthrough paths, while the Dashed Orange Arrows represent the taken paths where a conditional branch instruction can change the flow to a target block.

3.4.1 CFG Analysis

The features extracted in this analysis are based on the methodology established by Calder et al. in their foundational paper, "Evidence-Based Static Branch Prediction Using Machine Learning." [5]
For every single conditional branch, our pipeline looks at three things: the block holding the branch instruction itself (using the Br prefix), the target block if the condition is true and taken (using the T prefix), and the target block if the condition is false and falls through (using the F prefix). Because the structural properties on the True side apply exactly the same way to the False side, we measure the same identical metrics for both paths.

#	Feature Name	Description
1	`Br_LoopHead`	True if the basic block containing the branch instruction acts as the entry header of a loop.
2	`T_Dominates`	True if taken successor dominates the branch basic block.
3	`T_Postdominates`	True if the taken successor post-dominates the branch block.
4	`T_LoopHead`	True if the taken successor itself is identified as a loop header.
5	`T_Backedge`	True if the edge leading to the taken successor is a back-edge.
6	`T_Endswith`	Instruction Opcode that terminates the taken successor.
7	`T_LoopExit`	True if the edge leading to taken successor is a loop exit edge.
8	`T_HasCall`	True if the taken successor contains a `call` instruction.
9	`T_UseBefDef`	True if a Reg_Read register is consumed (used) inside the taken successor before it is defined.
10	`T_StoreToMem`	True if the taken successor contains a memory write instruction (e.g., a stack or heap store operation).
Note: Features 2 through 10 apply identically to the Fall through successor by replacing the T_ prefix with F_.		

Table 3.4: Features extracted from CFG

To extract the ten foundational features, a custom static analysis pipeline was implemented using the angr[2] framework, NetworkX, and PyVEX. The pipeline operates by reconstructing a fast, normalized Control Flow Graph (CFG) of the target function and mapping individual assembly blocks to numerical representations.

In a Control Flow Graph G = (V, E), structural dominance plays a critical role in understanding execution dependencies between a conditional branch basic block and its subsequent execution paths. Formally, a node d dominates a node n (denoted as d dom n) if every execution path originating from the function entry node to node n must pass through d. Based on the definitive compiler foundations established in "The Dragon Book" (Aho et al., 2006)[1], two core axioms govern

this relationship: every node inherently dominates itself, and the primary entry node dominates all other reachable nodes within the graph.

The Dominator Tree represents the dominance relationships in a graph, starting from the entry node where the parent of each node is its immediate dominator. To determine whether a node *d* dominates a node *n* using *Dominator Tree* is to traverse backward from *n* to the root and if node *d* is visited then d dominates *n*.

Control FLow Graph : $G = (V, E)$

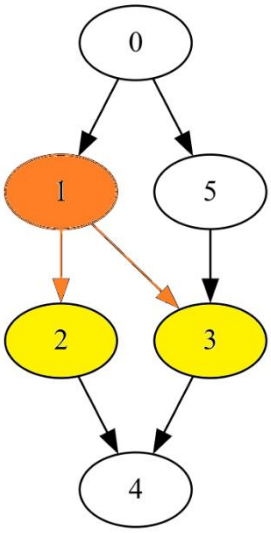
Dominator Tree : $T = (V, E^T)$

$E^T = \{ (idom(v), v) \mid \text{for all } v \in V, v \mathrel{!}= \text{entry node}\}$

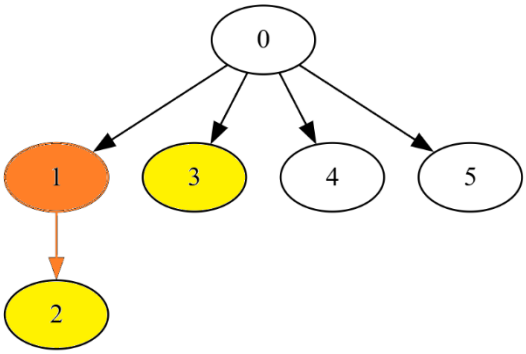
$Dominators(n) = \{ idom(n), idom(idom(n)), ..., \text{entry-node} \}$

If ($d \in Dominators(n)$) <=> d dominates n

Control Flow Graph



Dominator Tree



$Dom(0) = \{0\}$

$Dom(1) = \{0, 1\}$

$Dom(2) = \{0, 1, 2\}$

$Dom(3) = \{0, 3\}$

$Dom(4) = \{0, 4\}$

$Dom(5) = \{0, 5\}$

Node 1 is a conditional branch and its successors are {2, 3}.

$1 \in Dom(2)$, 1 Dominates 2

$1 \mathrel{!}\in Dom(3)$ 1 does not Dominates 3

Given a Control Flow Graph with an exit node, d post dominates n, if every path from n to the exit node includes d. Every node post dominates itself and the exit node post dominates every node in the Control Flow Graph.

Control Flow Graph : $G = (V, E)$

Reversed Graph : $GR = (V, ER)$

$ER = \{ (v, u) \mid (u, v) \in E\}$

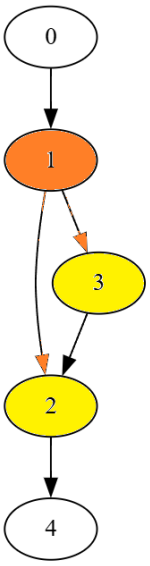
Dominator Tree : $T = (V, ET)$

$ET = \{ (idom(v), v) \mid \text{for all } v \in V, v \mathrel{!}= \text{entry node}\}$

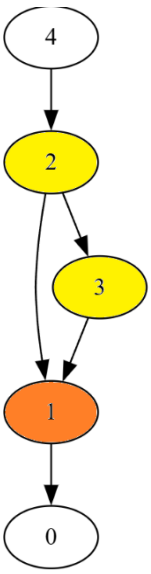
The Dominator Tree of the Reverse Graph shows whether a node d post dominates node n, if the path from n to the root of the tree includes d.

Dominance in Reversed CFG = Postdominance in CFG

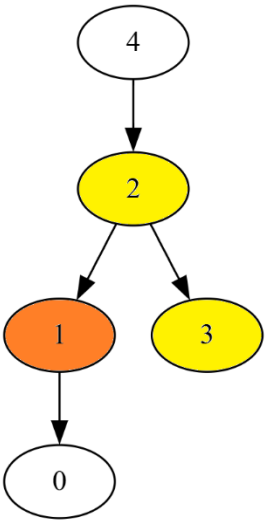
Control Flow Graph



Reversed CFG



D. Tree of the Rev. CFG



From the Reversed CFG:

$Dom(4) = \{4\}$

$Dom(2) = \{4, 2\}$

$Dom(1) = \{4, 2, 1\}$

$Dom(3) = \{4, 2, 3\}$

$Dom(0) = \{4, 2, 1, 0\}$

Node 1 is a conditional Branch with successor {2, 3}

$2 \in \mathrm{Dom}(1)$, 2 Dominates 1

$3 \notin \mathrm{Dom}(1)$, 3 does not Dominates 1

In conclude 2 post dominates 1 and 3 does not postdominates 1

Loops are present in most of the programs, and they are an important structure that helps the prediction of a conditional branch when it is a part of it

An edge is classified as a backedge, if there is an edge from a node d to a node n where n dominates d.

Control Flow Graph G = (V, E)

$(d,n) \in E$ AND $n \in$ Dominators(d) $\Leftrightarrow$ (d,n) is backedge

A node d in a Control Flow Graph is determined as a loop header if there is an edge from a node v to d and d dominates v.

Control FLow Graph G = (V, E)

$d \in V$, d is a backedge if:

$(n,d) \in E$ AND $d \in$ Dominators(n) $\Leftrightarrow$ d is a loop head

An edge in a Control FLow Graph is classified as a Loop Exit edge, if the source node is in a loop and the destination node is outside of the loop.

Control Flow Graph G =(V, E)

Edge $(d, v) \in E$

$l \subseteq V$, l be loop (set of nodes)

$l \in L$, L be the sets of nodes

$\exists\, l \in L$ such that $(d \in l) \wedge (v \notin l) \Leftrightarrow$ (d, v) loop exit edge

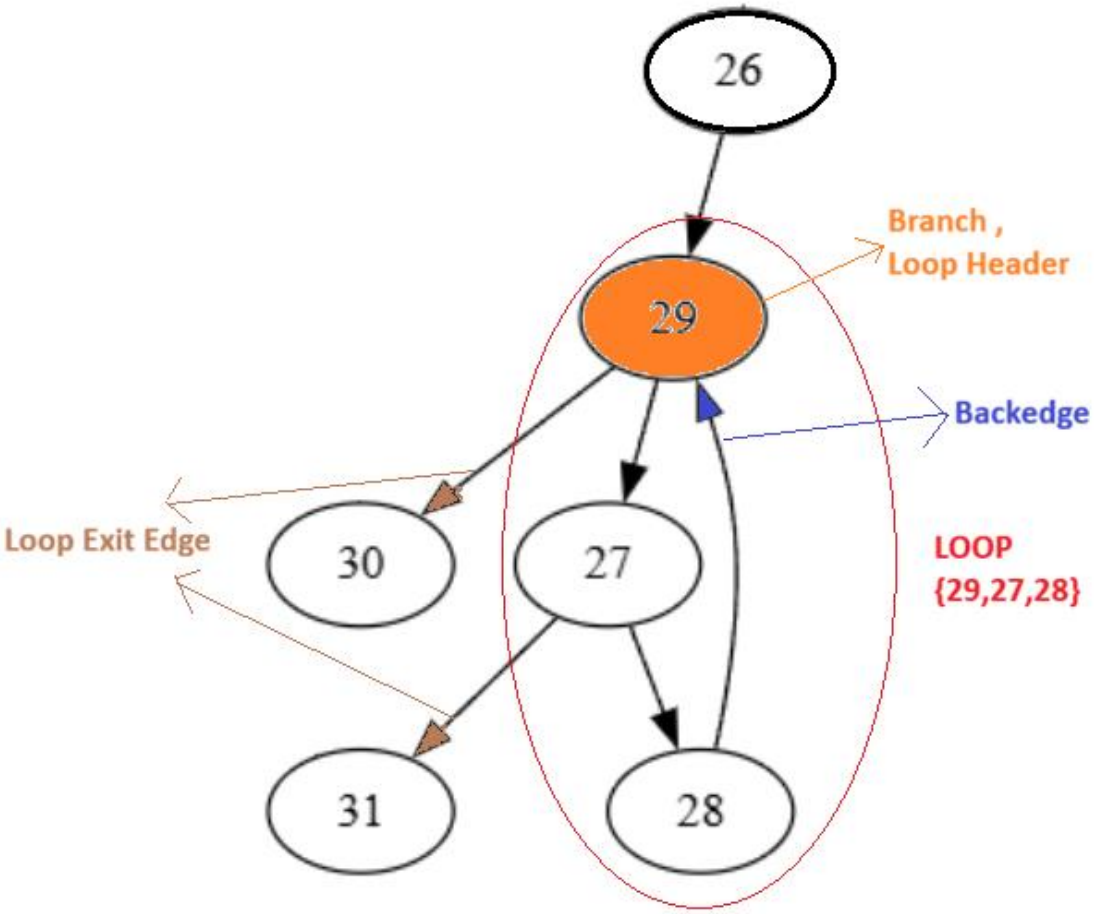


Figure 3.5: Loop representation in a CFG

Check whether the successor basic block of the branch ends with a CALL instruction

Branch Basic Block:

```
40113f:          push   %rbp
401140:    mov    %rsp,%rbp
401143:    sub    $0x10,%rsp
401147:    mov    %edi,-0x4(%rbp)
40114a:    cmpl   $0x1,-0x4(%rbp)
40114e:    jg     401157 <fact+0x18>
```

Fall Basic Block:

```
401150:    mov    $0x1,%eax
401155:    jmp    401168 <fact+0x29>
```

Taken Basic Block:

```
401157:     mov    -0x4(%rbp),%eax
40115a:     sub    $0x1,%eax
40115d:     mov    %eax,%edi
40115f:            callq  40113f <fact>
```

In this example the Fall successor does not contain a CALL instruction.
Taken successor contains a CALL instruction.

Having the set of registers read by the set_flag instruction and the basic block, UBD is true if there is a register that was read by an instruction and then defined in a different or the same instruction that was read by.

R = {r1, r2, …, rk} , R be set of registers read by the set flag instruction
r ∈ R , r be register
B = {i1, i2, …, ik}, B set of instructions in the basic block
i ∈ B , i be instruction
$\forall r \in R, UBD(B, r) = 1 \Leftrightarrow \exists ij , im \in B, j<=m : r \in Use(ij) \wedge r \in Def(im)$

Example:
Branch Basic Block

```
b1: cmpl   $0xa,-0x14(%rbp)
b2: je     4011b1
```

Taken Basic Block

```
t1: mov    -0x4(%rbp),%eax
t2: pop    %rbp
```

R = {rbp}
B ={t1, t2}
Use(t1) = {rbp, eax}
Def(t2) = {rbp}
UBD(B,rbp)=1

The store to memory information is retrieved while iterating the instruction of the successor and determining if the instruction writes to the memory.

Branch Basic Block:

```
40113f:     push   %rbp
401140:     mov    %rsp,%rbp
401143:     sub    $0x10,%rsp
401147:     mov    %edi,-0x4(%rbp)
40114a:     cmpl   $0x1,-0x4(%rbp)
40114e:     jg     401157 <fact+0x18>
```

Fall Basic Block:

```
401150:     mov    $0x1,%eax
401155:     jmp    401168 <fact+0x29>
```

Taken Basic Block:

```
401157:     mov    -0x4(%rbp),%eax
40115a:     sub    $0x1,%eax
40115d:     mov    %eax,%edi
40115f:     callq  40113f <fact>
```

The CALL instruction writes to memory

3.4.2 CFG Analysis Example

To validate the behavioral accuracy of the feature extraction pipeline, a structural case study was conducted using a sample binary compiled with compiler optimizations optimized for analysis (-O1 with function inlining disabled). The resulting Control Flow Graph (CFG) generated by the angr[2] visualization module is illustrated in Figure 3.6, and its corresponding extracted feature vector metrics are detailed in Table 3.5

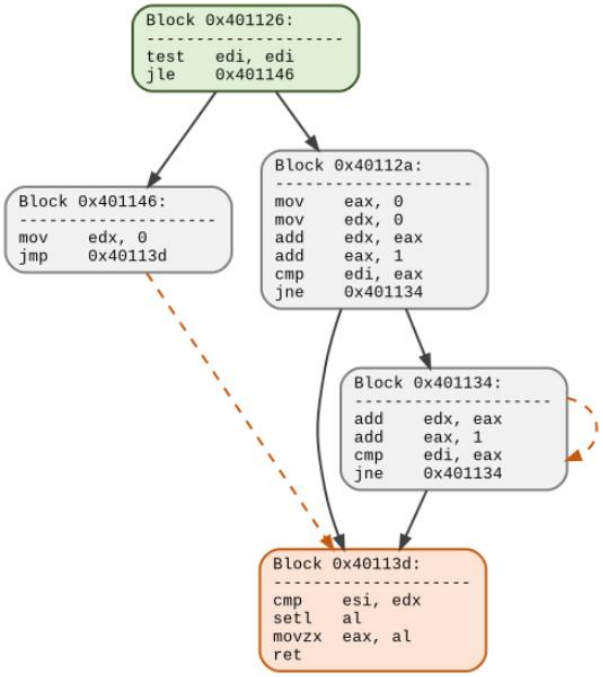


Figure 3.6: CFG created from angr

Feature Name	Instance 1 (0x40113b: JNZ)	Instance 2 (0x401128: JLE)
br_is_loop_header	1	0
t_dominates	1	1
t_post_dominates	1	0
t_is_loop_head	1	0
t_is_backedge	1	0
t_successor_ends	JNE	JMP
t_is_loop_exit	0	0
t_has_call	0	0
taken_ubd	1	0
taken_store	0	0
f_dominates	0	1
f_post_dominates	1	0
f_is_loop_head	0	0
f_is_backedge	0	0
f_successor_ends	RET	JNE
f_is_loop_exit	1	0
f_has_call	0	0
fall_ubd	0	0
fall_store	0	0

Table 3.5: Feature values extracted from angr

The first instance analyzed covers the cyclic loop-tracking branch (jnz 0x401134) within basic block 0x401134. The pipeline accurately marks br_is_loop_header as 1, matching the visual graph topology where this block acts as an active loop head. Because by taking the true branch routes execution directly back into its own starting address, the engine registers a self-loop, resolving t_is_loop_head and t_is_backedge to 1. When the loop terminates on a false condition, control flow escapes the loop structure directly into the terminal block 0x40113d (f_is_loop_exit = 1), which correctly reflects a RET instruction mnemonic. Since block 0x40113d functions as the absolute, unified return site under this optimization model, both t_post_dominates and f_post_dominates resolve to 1.

The second instance evaluates the independent guard condition (jle 0x401146) situated at the entry block 0x401126. As a pre-loop decision node, it naturally returns 0 for br_is_loop_header. Taking the true path routes control along the left bypass block 0x401146, which terminates with an unconditional jump (t_successor_ends = JMP) and hooks straight into the single exit node 0x40113d. Because this guard sits entirely outside the loop scope, it serves as a structural loop bypass rather than a loop exit node, assigning 0 to t_is_loop_exit. Symmetrically, the false path falls through to block 0x40112a (f_successor_ends = JNE) . Because these two execution paths diverge completely and represent mutually exclusive control corridors, neither target block post-dominates the branch node, correctly resolving both t_post_dominates and f_post_dominates to 0.

Chapter 4

Machine Learning Model

4.1 Introduction

The primary objective of this thesis is to accurately predict branch behaviors specifically their taken rates and directional biases by utilizing static binary characteristics and control flow graph (CFG) structures. Traditional architectural branch predictors rely heavily on runtime hardware history tables. While highly effective, these hardware mechanisms lack a global context of the application's overall control structure and require substantial hardware overhead. By shifting this predictive task into the software domain using static analysis and machine learning, it becomes possible to capture complex, non-local execution patterns without modifying hardware architecture.

To achieve this, the branch prediction task is modeled as a supervised multi-class classification problem. Rather than predicting raw execution paths directly, branches are categorized based on their historical behavior profiles across varying workloads. The predictive pipeline is built around a Deep Multilayer Perceptron (MLP) architecture implemented in PyTorch. The model acts as an analytical bridge: it takes abstract, static machine code attributes (such as register dependencies, local instruction sequences, and structural graph properties extracted via static analysis tool angr[2]) and maps them to dynamic behavioral classifications obtained via dynamic binary instrumentation (Intel Pin) [7].

The following sections detail the complete structural lifecycle of this machine learning framework. Section 4.1 describes the data engineering pipeline, focusing on the categorical classification threshold logic, dataset distribution layout, and

feature normalization techniques. Section 4.2 presents the evolutionary progression of the MLP network architecture, detailing the transition from primitive label encodings to continuous, high-density trainable embedding layers. Section 4.3 defines the exact optimization hyperparameters, regularizes, and reproducibility configurations, while Section 4.4 establishes the mathematical foundations of the evaluation metrics used to measure model accuracy and generalizability.

4.2 Data Processing

To build a reliable machine learning model for branch prediction, we must first clean and prepare our dataset. Raw data collected from static binary analysis and dynamic execution cannot be fed directly into a neural network. This section explains how we calculate our training labels, split the dataset across different benchmarks, clean up missing values, and extract input features from the code.

The raw data gives us two dynamic counters for each branch instruction i: Taken T_i, which is the number of times a branch was followed, and Executed E_i, which is the total number of times the branch was run. The continuous branch taken rate, denoted as R_i, is calculated using the formula:

$$R_i = \frac{T_i}{E_i}, \quad \text{where } E_i > 0 \text{ and } R_i \in [0, 1]$$

Instead of predicting the exact rate as a continuous number, we convert this task into a 3-class classification problem. For this labeling assignment, I set the highly biased rate threshold t at 0.005

Each branch is assigned a discrete class label Y_i in {0, 1, 2} based on the following mathematical logic:

$$Y_i = \begin{cases} 0 & \text{(Highly Not Taken / HNT)} \quad \text{if } R_i < \tau \\ 1 & \text{(Not Biased / NB)} \quad \text{if } \tau \leq R_i \leq 1-\tau \\ 2 & \text{(Highly Taken / HT)} \quad \text{if } R_i > 1-\tau \end{cases}$$

Class 0 (HNT): Branches that almost always fall through.
Class 2 (HT): Branches that are almost always taken.
Class 1 (NB): Dynamic, data-dependent branches that switch frequently and have no clear bias.

If we mix rows from the same program into both the training and testing sets, the model might simply memorize that program's specific layout instead of learning real, general software behaviors. To prevent this, we enforce a strict cross-validation split using separate, isolated workloads from the SPEC CPU2017 benchmark suite.

The complete dataset consists of 240,846 individual branch rows, which are divided into two globally isolated partitions:

Training Partition: Consists of 219,500 distinct rows extracted across 17 independent workloads: 500.perlbench_r, 502.gcc_r, 505.mcf_r, 507.cactuBSSN_r, 508.namd_r, 510.parest_r, 511.povray_r, 519.lbm_r, 523.xalancbmk_r, 525.x264_r, 526.blender_r, 527.cam4_r, 531.deepsjeng_r, 538.imagick_r, 541.leela_r, 544.nab_r, and 554.roms_r.
Testing Partition: Consists of 21,346 completely separate rows extracted from 5 entirely unseen workloads withheld during the training phase: 503.bwaves_r, 520.omnetpp_r, 548.exchange2_r, 549.fotonik3d_r, and 557.xz_r.

To guarantee that the machine learning model learns genuine software behavior patterns rather than cheating via easy math shortcuts, strict data leakage boundaries are established. Absolute execution metrics and unique layout memory markers are stripped completely from the input features before training begins. Specifically, the following columns are dropped: Taken, Executed, Regs_Read, Regs_Write, branch_bb_addr, taken_bb_addr, fall_bb_addr, Address, and Flag_Write_PC. Removing Taken and Executed prevents the model from reverse-engineering the target rate, while removing raw virtual

memory addresses forces the model to look at instruction behaviors instead of memorizing specific memory maps.

Conditional jumps are highly dependent on the local instruction sequence and register data flow immediately surrounding the branch. Because raw features are extracted as text components, string parsing operations are used to isolate precise dependencies.

The tracking fields Regs_Read and Regs_Write capture the register dependency tracking matrix. For register reads, a regular expression sequence isolates read operations to extract up to three individual read registers (reg1, reg2, reg3) alongside the specific opcodes that last updated them (reg1_Op, reg2_Op, reg3_Op). If the branch has fewer than three read dependencies, a placeholder token of "-1" pads out the empty vector slots. A parallel parsing sequence separates Regs_Write into three destination hardware registers (wreg1, wreg2), utilizing equivalent "-1" padding tokens to enforce uniform vector alignment.
For function-level layout tracking, the text labels in the Routine_Type column are converted directly into clean numerical assignments using a deterministic mapping function:

$$f(\text{Routine\_Type}) = \begin{cases} 0 & \text{if Routine\_Type} = \text{"NonLeaf"} \\ 1 & \text{if Routine\_Type} = \text{"Leaf"} \\ 2 & \text{if Routine\_Type} = \text{"Recursive"} \end{cases}$$

The engineered dataset tracks a complete set of features broken down into four functional microarchitectural groups. Across this thesis development, three distinct data normalization approaches were explored for these features before introducing them into the model architecture:

Feature Group	Variable Identifiers
Data-Flow Registers (5 Categorical Columns)	`reg1, reg2, reg3, wreg1, wreg2`

Instruction Opcodes (20 Categorical Columns)	`Opcode, Flag_Instr_Opcode`
	`Prev_Op_1 ... Prev_Op_5`
	`Next_Op_1 ... Next_Op_5`
	`reg1_Op, reg2_Op, reg3_Op`
	`t_successor_ends, f_successor_ends`

Graph Topology (19 Numerical Columns)	`Routine_Type, Same_BBL, br_is_loop_header`
	`t_dominates, f_dominates, t_post_dominates, f_post_dominates`
	`t_is_loop_head, f_is_loop_head, t_is_backedge, f_is_backedge`
	`t_is_loop_exit, f_is_loop_exit, t_has_call, f_has_call`
	`taken_store, fall_store, taken_ubd, fall_ubd`

Spatial Continuous (1 Continuous)	`Offset`

Table 4.1: Features in separate groups as handled for normalization

Categorical Label Encoding with StandardScaler (Models 1.1–1.3): Categorical features (Opcodes and Registers) are encoded via a LabelEncoder pipeline, which scans all values, finds unique string occurrences, sorts them alphabetically, and assigns sequential integers. Continuous numerical features are scaled using a StandardScaler to achieve a zero mean and unit variance via the equation:

$$x_{\text{new}} = \frac{x - \mu}{\sigma}$$

Categorical One-Hot Encoding (Model 1.4): To prevent the model from assuming false numerical relationships between unrelated arbitrary integer codes, categorical keys are converted into high-dimensional binary vectors via a OneHotEncoder.

Trainable Continuous Embedding Layers (Model 1.6): Instead of static string transformations, categorical attributes are mapped to low-dimensional trainable continuous vectors. This enables the model to actively optimize and map structural code context (such as register similarities and opcode families) directly within a shared high-density vector space.

4.3 Model Architecture

The architectural development focused on finding the optimal configuration to capture complex branch behaviors from static binary features. Several variants of the Multilayer Perceptron (MLP) were evaluated, progressively increasing in depth and sophistication.

Early iterations (Models 1.1–1.3) utilized a standard MLP approach with variations in hidden layer dimensions and regularization. To combat overfitting and improve training stability, Batch Normalization was introduced in Model 1.3, alongside a reduction in dropout probability. Model 1.4 marked a significant shift by adopting One-Hot Encoding to handle high-cardinality categorical data, resulting in a substantially wider input layer. The final refined architecture, Model 1.6, moved away from sparse encodings entirely, employing trainable embedding layers. This allowed the model to learn dense, continuous vector representations for registers and opcodes, significantly improving the feature extraction capability of the network.

Model	Architecture Specification	Norm.
Standard MLP Architectures		
1.1	$L(59,128) \to BN \to ReLU \to D(0.3) \to L(128,64) \to BN \to ReLU \to D(0.3) \to L(64,3)$	LabelEnc.
1.2	$L(59,256) \to BN \to ReLU \to D(0.3) \to L(256,128) \to BN \to ReLU \to D(0.3) \to L(128,3)$	LabelEnc.
1.3	$L(59,256) \to BN \to ReLU \to D(0.1) \to L(256,128) \to BN \to ReLU \to D(0.1) \to L(128,3)$	LabelEnc.
1.4	$L(2395,256) \to BN \to ReLU \to D(0.2) \to L(256,128) \to BN \to ReLU \to D(0.2) \to L(128,3)$	OneHot
Advanced Embedding Architecture		
1.6	$\text{Embed} \mapsto L(606,512) \to BN \to ReLU \to D(0.4) \to L(512,256) \to BN \to ReLU \to D(0.4) \to L(256,3)$	Embed.

Table 4.2: Model architectures use

4.4 Training Configuration

The training pipeline for Model 1.6 was designed to maximize the efficacy of learnable embedding layers while ensuring robustness against overfitting.

To optimize the model, Cross-Entropy Loss was utilized to evaluate the performance of the 3-class classification objective (HNT, NB, HT). Mathematically, the loss for each sample is defined as the negative log-likelihood of the true class:

$$L = -\sum_{c=1}^{3} y_{o,c} \log(p_{o,c})$$

Where $y_{o,c}$ is a binary indicator (0 or 1) if class label c is the correct classification for observation o, and $p_{o,c}$ is the predicted probability of the model assigning observation o to class c. This function penalizes the model exponentially as the predicted probability deviates from the target label, forcing the network to minimize the divergence between the predicted probability distribution and the ground truth.

Because the branch prediction dataset exhibits an inherent class imbalance, the loss function was weighted by the inverse frequency of each class. This adjustment ensures that misclassifications of minority classes are penalized more severely, preventing the model from achieving high accuracy by simply predicting the majority class. The model was optimized using the Adam optimizer with a learning rate of 0.001 and a weight decay of 0.0001 to promote regularization. All models were trained for a maximum of 150 epochs using a batch size of 512, with an early stopping mechanism (patience of 30 epochs) and a globally fixed random seed of 1.

Hyperparameter	Value / Configuration
Loss Function	Cross-Entropy (Inverse-Frequency Weighted)
Embedding Dimension	24
Optimizer	Adam (lr = 0.001)
Weight Decay	0.0001
Batch Size	512
Max Epochs	150
Early Stopping Patience	30 epochs
Random Seed	1

Table 4.3: Configurations of Model 1.6

4.5 Evaluation Metrics

To rigorously assess the performance of Model 1.6, we utilize Accuracy and the Macro-average F1-Score, derived from the confusion matrix. These metrics allow us to evaluate both the overall predictive power and the classification balance across the HNT, NB, and HT classes.

Precision: Measures the accuracy of positive predictions (e.g., when the model predicts HT, how often is it actually HT?). High precision indicates a low false-positive rate.
Recall: Measures the model's ability to find all actual instances of a class (e.g., of all true HT branches, how many did the model correctly identify?).
F1-Score: The harmonic mean of precision and recall. It balances these two metrics, providing a single performance indicator that is particularly useful for assessing models where class distributions might be skewed.

The confusion matrix below illustrates the model's predictive patterns across the 21,351 test samples. The rows represent the actual ground truth, while the columns represent the model's predictions.

	Pred: HNT	Pred: NB	Pred: HT
True: HNT	8123	1296	812
True: NB	1039	3974	604
True: HT	812	1081	3610

Table 4.4: Confusion Matrix results of Model 1.6

The matrix reveals how the model distinguishes between branch types:

HNT Reliability: The model shows high confidence in identifying HNT branches (8123 correct). The off-diagonal errors (1296 predicted as NB and 812 as HT) suggest that while the model is robust, it occasionally mislabels "Not Taken" branches as dynamic or "Taken" branches.
NB Characteristics: The model correctly identifies 3,974 NB branches. However, 1,039 NB instances are incorrectly labeled as HNT, suggesting that some dynamic branches share enough static feature similarities with falling-through branches to cause confusion.
HT Identification: The model identified 3,610 HT branches correctly. The errors (1,081 instances misclassified as NB) indicate that the model finds it difficult to distinguish highly biased "Taken" branches from branches that behave dynamically in certain edge cases

Model 1.6 achieved an overall accuracy of 0.736 and a macro-averaged F1-score of 0.718. The model's architecture, specifically the use of 24-dimensional embedding layers, allows it to learn dense representations of opcode and register patterns that static encoding methods cannot replicate. The balanced macro-F1 score relative to the raw accuracy confirms that the model is not merely biasing its predictions toward the majority class (HNT), but is successfully learning the underlying microarchitectural features required to distinguish between biased and non-biased dynamic code patterns.

Chapter 5

Experimental Results and Analysis

5.1 Introduction

This chapter evaluates the predictive performance and generalization capabilities of Model 1.6 across the SPEC CPU2017 benchmark suite. The experimental framework employs a Leave-One-Benchmark-Out (LOO) methodology, systematically withholding one workload during training to assess the model's ability to maintain high classification accuracy on unseen execution patterns. This evaluation is critical for determining how effectively the model captures generalizable microarchitectural features versus overfitting to the specific biases of a training set. Furthermore, this chapter provides a feature-importance analysis to isolate the influence of specific instruction-level attributes such as opcode sequences and register dependencies on overall prediction stability. By analyzing both the peak performance benchmarks and the most challenging workloads, we provide a detailed explanation of the structural factors that govern branch predictability in modern software.

5.2 Leave-One-Benchmark-Out evaluation

The Leave-One-Benchmark-Out (LOO) evaluation serves as the primary stress test for our branch prediction model. In this setup, we iteratively train the model on the full SPEC CPU2017 suite while excluding one specific workload, which is then used as a strictly unseen validation set. The variance in accuracy across benchmarks allows us to characterize the model's performance on different types of computational tasks, ranging from high-predictability scientific simulations to complex, irregular general-purpose applications.

The following table details the predictive accuracy and macro F1-score for each benchmark, ranked by performance.

Benchmark	Accuracy	Macro F1	Test Samples
519.lbm_r	0.899	0.877	1,023
503.bwaves_r	0.882	0.867	2,274
549.fotonik3d_r	0.819	0.766	4,752
554.roms_r	0.793	0.733	6,838
541.leela_r	0.773	0.770	4,077
544.nab_r	0.754	0.730	2,961
548.exchange2_r	0.748	0.734	3,267
508.namd_r	0.733	0.720	3,504
527.cam4_r	0.709	0.668	23,068
557.xz_r	0.703	0.674	2,128
538.imagick_r	0.685	0.643	4,052
520.omnetpp_r	0.675	0.657	8,930
511.povray_r	0.669	0.638	7,297
510.parest_r	0.666	0.638	15,557
525.x264_r	0.644	0.631	4,480
531.deepsjeng_r	0.633	0.635	2,629
507.cactuBSSN_r	0.632	0.600	11,586
523.xalancbmk_r	0.588	0.550	13,595
526.blender_r	0.575	0.551	14,864
505.mcf_r	0.570	0.549	309
500.perlbench_r	0.523	0.498	17,137
502.gcc_r	0.504	0.475	86,513

Table 5.1: LOO Evaluation Results per Benchmark

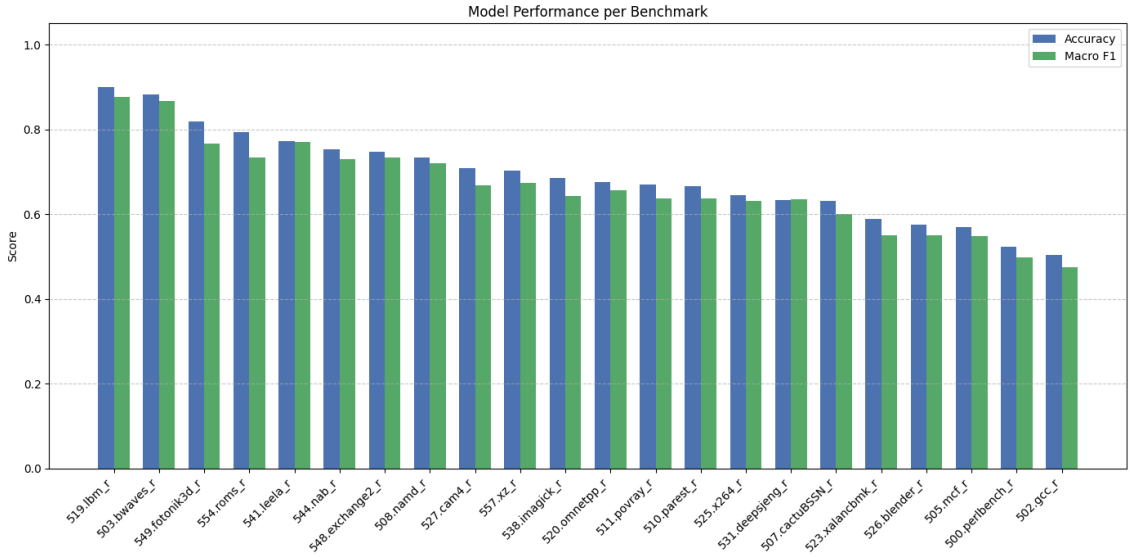


Figure 5.1: Representation of accuracy and F1 score results in LOO method

Benchmarks such as 519.lbm_r and 503.bwaves_r achieve the highest accuracy (above 88%). These are typically scientific applications dominated by predictable, high-count loop iterations. The model's ability to predict these with high fidelity suggests that the embedding layers have successfully captured the underlying regularity of these compute-heavy patterns.

The model encounters significant predictive degradation in integer-heavy workloads like 502.gcc_r and 500.perlbench_r. These benchmarks contain the largest number of test samples and represent general-purpose software with complex, state-dependent branching.

5.2.1 Analysis of Highest Performance Benchmarks based on Accuracy

The predictive performance across benchmarks reveals that the model's efficacy is inherently tied to the software's control-flow structure. While scientific simulations consistently show high predictability, general-purpose applications introduce complexities that challenge static-feature models. To examine these limits, we compare the top-performing benchmarks against two specific cases that yielded lower accuracy

Focusing on the high-performing 519.lbm_r (Accuracy: 0.899, F1: 0.877), the model demonstrates strong discriminative capabilities across the defined branch categories: Highly Not Taken (HNT), Non-Biased (NB), and Highly Taken (HT).

	Predicted HNT	**Predicted NB**	**Predicted HT**
Actual HNT	476	29	8
Actual NB	23	151	11
Actual HT	10	22	293

Table 5.2: Confusion Table of lbm

The matrix confirms that the model maintains sharp decision boundaries for biased branches. With 476 of 513 HNT samples and 293 of 325 HT samples correctly classified, the model demonstrates high sensitivity to steady-state execution patterns. The primary source of misclassification is the NB category, which accounts for the majority of off-diagonal entries.

Opcode	**Total**	**Mispredictions**	**Error Rate (%)**
JZ	493	44	8.92%
JNZ	271	27	9.96%
JBE	61	9	14.75%
JNBE	52	10	19.23%
JLE	43	2	4.65%

Table 5.3: Most mispredictions based on Opcode in lbm

As shown in Table 5.3, the model maintains a high degree of accuracy for the most frequent jump instructions, JZ (8.92% error rate) and JNZ (9.96% error rate). The higher error rates observed in less frequent instructions like JNBE (19.23%) suggest that the model encounters greater difficulty in resolving patterns for sparse, complex conditional transitions.

This error distribution reinforces the conclusion that the model's performance is tightly coupled with structural regularity. By successfully predicting nearly 90% of the branch outcomes in a scientific workload, the model demonstrates that it has effectively learned the core iterative patterns of fluid dynamics. The remaining 10% of errors are localized within the least frequent, more stochastic branching instructions, marking the natural boundary of the current architecture's predictive capabilities.

The same work happened on bwaves_r benchmark.

The 503.bwaves_r benchmark presents a varied workload, resulting in an accuracy of 0.882 and a Macro F1-score of 0.867. While the model remains effective, the distribution of errors across opcode types provides deeper insight into the model's limitations in higher-entropy environments.

	Predicted HNT	**Predicted NB**	**Predicted HT**
Actual HNT	1063	84	26
Actual NB	42	470	28
Actual HT	16	72	473

Table 5.4: Most mispredictions based on Opcode in bwaves

The confusion matrix highlights a clear performance disparity regarding the NB (Non-Biased) class. While the model maintains high accuracy for biased branches (HNT and HT), the recall for the NB category is notably impacted. With 84 HNT branches and 72 HT branches being incorrectly mapped to the NB

category, it is evident that the model faces greater difficulty in resolving the state of non-biased branches within this workload. The NB category consistently yields the lowest F1-score across all tested benchmarks, acting as a focal point for misclassification where branch behavior deviates from steady-state iterative patterns.

Opcode	Total	Mispredictions	Error Rate (%)
JZ	930	120	12.90%
JNZ	661	78	11.80%
JBE	114	20	17.54%
JLE	206	17	8.25%
JNBE	105	10	9.52%

Table 5.5: Most mispredictions based on Opcode in bwaves

As shown in Table 5.5, the model maintains a relatively low error rate for the most frequent jump instructions, particularly JZ (12.90%) and JNZ (11.80%). This indicates that despite the increased variety in the instruction stream, the model effectively maps most of the bwaves_r control flow to predictable outcomes. The higher error rate in JBE (17.54%) suggests that this specific jump type captures the limits of the model's predictive resolution within this scientific simulation.

These results suggest that the NB category represents a structural threshold for the model. While it successfully captures the "rhythm" of highly biased loops, the precision decreases when encountering the heightened entropy of general-purpose conditional branching found in bwaves_r. This confirms that the model’s performance is limited by the stochastic nature of data-dependent jumps, particularly when the ratio of NB branches increases.

5.2.2 Analysis of Lowest Performance Benchmarks based on Accuracy

Analysis of Lowest Performance starts with perlbench. Unlike scientific simulations that rely on iterative, structured control flow, 500.perlbench_r is characterized by highly irregular, data-dependent branching. This complexity resulted in an accuracy of 0.523 and a Macro F1-score of 0.498. The confusion matrix below quantifies the model's struggle to maintain classification stability:

	Predicted HNT	Predicted NB	Predicted HT
Actual HNT	5002	1727	1141
Actual NB	1819	2026	915
Actual HT	1447	1126	1934

Table 5.6: Confusion Table of perlbench

The matrix demonstrates a significant loss of discriminative power. The model frequently misclassifies biased branches (HNT/HT) as NB, indicating that the "Non-Biased" category is no longer functioning as a transitionary buffer, but rather as an area of high uncertainty for the model.

Metric	Value
Total Branches	17,137
Total Mispredictions	8,175 (47.7%)
Loop-Related Mispredictions	334 (4.09%)

Table 5.7: Mispredictions realtedd to br_LoopHeader

The most significant finding is that loop-related branches account for only 4.09% of all mispredictions. This implies that the model's accuracy is not degrading because it fails to identify iterative structures—it is degrading because the vast majority of 500.perlbench_r is composed of unpredictable, non-iterative conditional logic. The model retains its ability to identify loops where they exist, but those instances are statistically overwhelmed by the high-entropy nature of the remaining instruction stream. This performance gap validates the hypothesis that the proposed model excels in structured scientific environments but

encounters a "complexity wall" when navigating the non-deterministic control flow typical of general-purpose applications.

Opcode	Total	Mispredictions	Error Rate (%)
JZ	8989	4507	50.14%
JNZ	4931	2261	45.85%
JBE	767	376	49.02%
JNBE	608	247	40.63%
JNB	493	202	40.97%

Table 5.8: Most mispredictions based on Opcode in perlbench

The granular analysis in Table 5.8 reveals that for the primary opcodes (JZ, JNZ, JBE), the model consistently mispredicts roughly 40–50% of instances.

The 505.mcf_r benchmark represents a specific challenge for the model: combinatorial optimization. Unlike the previous benchmarks, mcf_r is characterized by a high volume of memory-intensive operations and pointer-chasing, leading to a control-flow pattern that is more sparse but highly sensitive to data-dependent conditions.

The model achieves an accuracy of 0.570 and a Macro F1-score of 0.549 on 505.mcf_r. While this represents a modest improvement over perlbench_r, it highlights a continued difficulty in distinguishing between biased and non-biased branches within memory-heavy workloads.

	Predicted HNT	Predicted NB	Predicted HT
Actual HNT	68	49	15
Actual NB	23	82	7
Actual HT	8	31	26

Table 5.9: Confusion Table of mcf

The confusion matrix shows that the model successfully identifies the majority of NB branches (82/112). However, it struggles with the high-bias HT category, frequently misclassifying these branches as NB. This suggests that in mcf_r, many branches that exhibit high-bias behavior are structurally masked by complex memory-dependent lookups, causing the model to default to the NB classification.

Opcode	Total	Mispredictions	Error Rate (%)
JZ	85	33	38.82%
JNZ	61	32	52.46%
JLE	42	18	42.86%
JS	35	14	40.00%
JNLE	23	11	47.83%

Table 5.10: Most mispredictions based on Opcode in mcf

Table 10 demonstrates that the model maintains a relatively consistent error profile across the most frequent jump types, with the JNZ opcode showing a particularly high error rate of 52.46%.

Metric	Value
Total Branches	309
Total Mispredictions	133 (43.04%)
Loop-Related Mispredictions	22 (16.54%)

Table 5.11: Mispredictions based on br_LoopHead mcf

In contrast to 500.perlbench_r, the loop-related misprediction rate in 505.mcf_r is significantly higher at 16.54%. This is a crucial observation: the model's errors are more closely tied to iterative structures here than in previous general-purpose benchmarks. This indicates that mcf_r possesses more coherent loop structures, but the model struggles to correctly classify the entry/exit points of these loops

due to the underlying memory-access latency, which interferes with the static features derived from instruction sequences.

5.3 Feature-Importance Analysis

To evaluate the contribution of individual feature groups to the model's predictive accuracy, an ablation study was conducted. By systematically removing specific metadata, the sensitivity of the MLP to various program features was quantified. The baseline configuration achieved an F1-score of 0.713 and an accuracy of 0.732.

Configuration	F1-score	Accuracy	Drop
Baseline	0.713	0.732	0.000
no_loop	0.708	0.728	0.005
no_dominance	0.711	0.731	0.001
no_call	0.707	0.725	0.006
no_store_ubd	0.710	0.728	0.003
no_size	0.711	0.729	0.002
no_offset	0.711	0.729	0.001
no_context_opcodes	0.695	0.714	0.017
no_registers	0.699	0.719	0.014

Table 5.12: Drop of accuracy and F1-score when a group is out

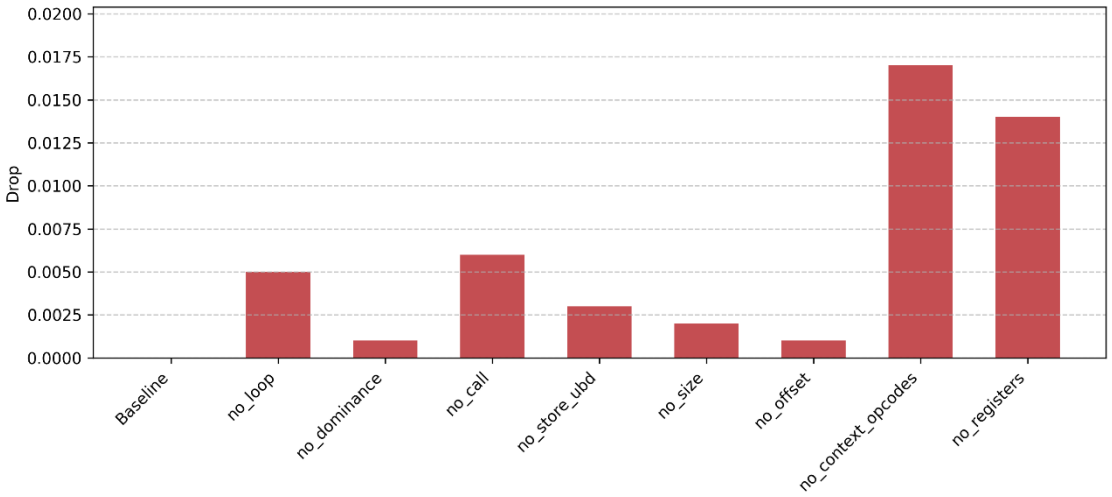


Figure 5.1: Graph representing the drop of accuracy

The results indicate that the model's performance is most significantly dependent on local and functional context. The largest performance degradation occurred upon the removal of opcode context (no_context_opcodes), resulting in a 0.017

drop in macro F1-score. This underscores the necessity of analyzing surrounding instructions to capture the "rhythm" of the execution flow.

The second most impactful feature group was register usage (no_registers), which caused a 0.014 drop in F1-score. This highlights the importance of data-flow information, the model clearly relies on register-state metadata to distinguish between different logical paths when structural markers (like loops or dominance) are insufficient.

Conversely, structural features such as no_dominance and no_offset had negligible effects on performance. This suggests that while control-flow graph (CFG) metadata is useful for static analysis, it is secondary to the functional behavior represented by the opcode and register embeddings. In effect, the model achieves high accuracy by prioritizing the "behavioral" context of a branch over its "topological" placement within the program.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The challenge of branch misprediction remains a critical performance bottleneck in modern processor architectures. As program complexity increases, traditional dynamic history-based predictors face diminishing returns, particularly when encountering cold-start scenarios or irregular branch patterns. This thesis investigated whether a machine learning-based approach, trained on static instruction-level and structural program data, could successfully identify and classify highly biased conditional branches, those that consistently resolve to a single outcome. The motivation for this research was to determine if the "rhythm" of a program, its static opcode sequence and register-dependency patterns, contains sufficient information to isolate these highly predictable branches without requiring extensive runtime execution history.

Experimental results, conducted across a diverse set of SPEC CPU2017 benchmarks, confirmed that branch predictability is not merely a dynamic property, but one deeply embedded in the static structure of code. A central finding is that a substantial proportion of conditional branches exhibit a high degree of bias, effectively acting as "near-constant" paths within the execution flow. While scientific simulations like lbm_r and bwaves_r demonstrated high structural regularity and strong predictive outcomes for these biased branches, general-purpose applications like perlbench_r introduced significantly higher entropy. This revealed that the model’s efficacy in isolating highly biased branches is bounded by the nature of the application. However, even in high-entropy scenarios, the model consistently identified key operational patterns through opcode and register embeddings, proving that static features provide a unique perspective on branch bias that dynamic-only predictors often overlook.

The significance of this research lies in the potential to fundamentally shift the methodology of branch prediction by proactively isolating highly biased branches. By demonstrating that such branches can be classified through static analysis, this work provides a viable mechanism to mitigate the performance penalties associated with pipeline stalls and cold-start flushes. The ablation study further proved that functional context, such as local opcode sequences, is a more robust predictor of bias than traditional structural graph metrics. This confirms that localized functional context provides a clearer "intent" of the program's execution flow and a more reliable indicator of branch bias than abstract topological features.

Ultimately, this work serves as a bridge between static binary analysis and high-performance hardware execution. By transforming static program knowledge into actionable intelligence regarding branch bias, this thesis demonstrates that the future of processor efficiency may lie in a hybrid approach: one that combines the speed of dynamic prediction with the proactive, pattern-aware insights provided by machine learning. This dual approach offers an immediate path to reducing branch-induced latency by bypassing unnecessary dynamic lookups for highly biased branches, effectively optimizing the processor for the code it executes.

6.2 Future work

While this thesis provides a comprehensive analysis of branch predictability using both dynamic and structural features, there remain several avenues for further investigation. One promising direction is the integration of sequence-aware models such as recurrent neural networks or Transformer-based architectures. These models could capture temporal dependencies in instruction sequences and loops more effectively, potentially improving predictions for branches that are highly context-dependent or occur within nested loops.

The application of this methodology to real-time or hardware-integrated branch prediction represents another important direction. By embedding predictive

models directly within CPU architectures or compiler pipelines, it may be possible to reduce pipeline stalls and optimize execution without relying solely on static hardware predictors. This would require careful consideration of computational efficiency and latency constraints.

Finally, expanding the evaluation to include multi-core processors, out-of-order execution, and cross-benchmark generalization could provide a deeper understanding of branch behavior under more realistic conditions. Such studies would not only enhance the practical applicability of the model but also provide insights that could inform compiler optimizations and architectural design decisions in future processor generations.

References

[1] Abo, Alfred, et al. *Second Edition*.

https://faculty.sist.shanghaitech.edu.cn/faculty/songfu/cav/Dragon-book.pdf

[2] "Angr." *GitHub*, 20 May 2026, github.com/angr. Accessed 20 May 2026.

https://github.com/angr

[3] Antoniou, Georgia, et al. "Agile C-States: A Core C-State Architecture for Latency Critical Applications Optimizing Both Transition and Cold-Start Latency." *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 4, 19 Nov. 2024, pp. 1–26,

https://doi.org/10.1145/3674734. Accessed 20 May 2026.

[4] Ball, Thomas, and James R. Larus. "Branch Prediction for Free." *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 300–313,

https://doi.org/10.1145/155090.155119. Accessed 20 May 2026.

[5] Calder, Brad, et al. "Evidence-Based Static Branch Prediction Using Machine Learning." *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, Jan. 1997, pp. 188–222, cseweb.ucsd.edu/~calder/papers/TOPLAS-97-ESP.pdf,

https://doi.org/10.1145/239912.239923.

[6] Mcfarling, Scott. *Combining Branch Predictors*. 1993.

http://classweb.ece.umd.edu/enee646.F2007/combining.pdf

[7] "Pin: Pin 3.21 User Guide." *Intel.com*, 2021,

software.intel.com/sites/landingpage/pintool/docs/98484/Pin/html/index.html.